Lecture No. 23

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill

Overview of Lecture

Physical Record and De-normalization

Partitioning

In the previous lecture, we have studied different data types and the coding techniques. We have reached now on implementing our database in which relations are now normalized. Now we will make this database efficient form implementation point of view.

Physical Record and Denormalization

Denormalization is a technique to move from higher to lower normal forms of database modeling in order to speed up database access. Denormalization process is applied for deriving a physical data model from a logical form. In logical data base design we group things logically related through same primary key. In physical database design fields are grouped, as they are stored physically and accessed by DBMS. In general it may decompose one logical relation into separate physical records, combine some or do both. There is a valid reason for denormalization that is to enhance the performance. However, there are several indicators, which will help to identify systems, and tables, which are potential denormalization candidates. These are:

Many critical queries and reports exist which rely upon data from more than one table. Often times these requests need to be processed in an on-line environment.

Repeating groups exist which need to be processed in a group instead of individually.

Many calculations need to be applied to one or many columns before queries can be successfully answered.

Tables need to be accessed in different ways by different users during the same timeframe.

Certain columns are queried a large percentage of the time. Consider 60% or greater to be a cautionary number flagging denormalization as an option.

We should be aware that each new RDBMS release usually bring enhanced performance and improved access options that may reduce the need for denormalization. However, most of the popular RDBMS products on occasion will require denormalized data structures. There are many different types of denormalized tables, which can resolve the performance problems caused when accessing fully normalized data. Denormalization must balance the need for good system response time with the need to maintain data, while avoiding the various anomalies or problems associated with denormalized table structures. Denormalization goes hand-in-hand with the detailed analysis of critical transactions through view analysis. View analysis must include the specification of primary and secondary access paths for tables that comprise end-user views of the database. A fully normalized database schema can fail to provide adequate system response time due to excessive table join operations

Denormalization Situation 1:

Merge two Entity types into one with one to one relationship. Even if one of the entity type is optional, so joining can lead to wastage of storage, however if two accessed together very frequently their merging might be a wise decision. So those two relations must be merged for better performance, which have one to one relationship.

Denormalization Situation 2:

Many to many binary relationships mapped to three relations. Queries needing data from two participating ETs need joining of three relations that is expensive. Join is an expensive operation from execution point of view. It takes time and lot of resources. Now suppose there are two relations STUDENT and COURSE and there exits a many to many relationship in between them. So there are three relations STUDENT, COURSE and ENROLLED in between them. Now if we want to see that a student has enrolled how many courses. So to get this we will have to join three relations, first the STUDENT and ENROLLED and then joining it with COURSE, which is quite expensive. The relation created against relationship is merged with one of the relation created against participating ETs. Now the join operation will be performed only once. Consider the following many to many relationship:-

EMP (empID, eName,pjId,Sal)
PROJ (pjId,pjName)
WORK (empId.pjId,dtHired,Sal)

This is a many to many relationship in between EMP and PROJ with a relationship of WORK. So now if we by de-normalizing these relations and merge the WORK relation with PROJ relation, which is comparatively smaller one. But in this case it is violating 2NF and anomalies of 2NF would be there. But there would be only one join operation involved by joining two tables, which increases the efficiency.

EMP (empID, eName,pjId,Sal)
PROJ (pjId,pjName, empId,dtHired,Sal)

So now it is up to you that you want to weigh the drawbacks and advantages of denormalization.

Denormalization Situation 3:

Reference Data: One to many situation when the ET on side does not participate in any other relationship, then many side ET is appended with reference data rather than the foreign key. In this case the reference table should be merged with the main table.

We can see it with STUDENT and HOBBY relations. One student can have one hobby and one hobby can be adopted by many students. Now in this case the hobby can be merged with the student relation. So in this case although redundancy of data would be there, but there would not be any joining of two relations, which will have a better performance.

Partitioning

De-normalization leads to merging different relations, whereas partitioning splits same relation into two. The general aims of data partitioning and placement in database are to

1. Reduce workload (e.g. data access, communication costs, search space)
2. Balance workload
3. Speed up the rate of useful work (e.g. frequently accessed objects in main memory)

There are two types of partitioning:-

Horizontal Partitioning

Vertical Partitioning

Horizontal Partitioning:
Table is split on the basis of rows, which means a larger table is split into smaller tables. Now the advantage of this is that time in accessing the records of a larger table is much more than a smaller table. It also helps in the maintenance of tables, security, authorization and backup. These smaller partitions can also be placed on different disks to reduce disk contention. Some of the types of horizontal partitioning are as under:-

Range Partitioning:
In this type of partitioning range is imposed on any particular attribute. So in this sway different partitions are made on the basis of those ranges with the help of select statement. For Example for those students whose ID is from 1-1000 are in partition 1 and so on. This will improve the overall efficiency of the database. In range partition the partitions may become unbalanced. So in this way few partitions may be overloaded.

Hash Partitioning:
It is a type of horizontal partitioning. In this type particular algorithm is applied and DBMS knows that algorithm. So hash partitioning reduces the chances of unbalanced partitions to a large extent.

List Partitioning:
In this type of partitioning the values are specified for every partition. So there is a specified list for all the partitions. So there is no range involved in this rather there is a list of values.
Summary:
De-normalization can lead to improved processing efficiency. The objective is to improve system response time without incurring a prohibitive amount of additional

data maintenance requirements. This is especially important for client-server systems. Denormalization requires thorough system testing to prove the effect that denormalized table structures have on processing efficiency. Furthermore, unseen ad hoc data queries may be adversely affected by denormalized table structures. Denormalization must be accomplished in conjunction with a detailed analysis of the tables required to support various end-user views of the database. This analysis must include the identification of primary and secondary access paths to data. Similarly before carrying out partitioning of the table thorough analysis of the relations is must.

Exercise:

Critically examine the tables drawn for Examination system and see if there is a requirement of denormalization and partitioning and then carry out the process.

Lecture No. 24

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill

Overview of Lecture

- Vertical Partitioning
- Replication
- Structured Query Language (SQL)

In the previous lecture we were discussing physical data base design, in which we studied denormalization and its different aspects. We also studied the horizontal partitioning. In this lecture we will study vertical partitioning.

Vertical Partitioning

Vertical partitioning is done on the basis of attributes. Same table is split into different physical records depending on the nature of accesses. Primary key is repeated in all vertical partitions of a table to get the original table. In contrast to horizontal partitioning, vertical partitioning lets you restrict which columns you send to other destinations, so you can replicate a limited subset of a table's columns to other machines. We will now see it with the example of a student relation as under: -

STD (stId, sName, sAdr, sPhone, cgpa, prName, school, mtMrks, mtSubs, clgName, intMarks, intSubs, dClg, bMarks, bSubs)

Now in this relation the student relation has number of attributes. It is in 3NF . But the nature of accesses in this relation is different. So now we will partition this relation vertically as under.

STD (stId, sName, sAdr, sPhone, cgpa, prName)

STDACD (sId, school, mtMrks, mtSubs, clgName, intMarks, intSubs, dClg, bMarks, bSubs)

Replication

The process of copying a portion of the database from one environment to another and keeping subsequent copies of the data in synchronization with the original source Changes made to the original source are propagated to the copies of the data in other environments. It is the final form of denormalization. It increases the access speed and failure damage of the database. In replication entire table or part of table can be replicated. Replication is normally adopted in those applications, where updation is not very frequent, because if updation is frequent so then it will have problems of updation in all the copies of database relations. This will also slow down the speed of database.

Clustering Files

Clustering is a process, which means to place records from different tables to place in adjacent physical locations, called clusters. It increases the efficiency since related records are placed close to each other. Clustering is also suitable to relatively static situations. The advantage of clustering is that while accessing the records it is easy to access. Define cluster, define the key of the cluster, and include the tables into the cluster while creating associating the key with it.

Summary of Physical Database Design

Database design is the process of transforming a logical data model into an actual physical database. A logical data model is required before you can even begin to design a physical database. The first step is to create an initial physical data model by transforming the logical data model into a physical implementation based on an understanding of the DBMS to be used for deployment. To successfully create a physical database design you will need to have a good working knowledge of the features of the DBMS including:

- In-depth knowledge of the database objects supported by the DBMS and the physical structures and files required to support those objects.
- Details regarding the manner in which the DBMS supports indexing, referential integrity, constraints, data types, and other features that augment the functionality of database objects.

- Detailed knowledge of new and obsolete features for particular versions or releases of the DBMS to be used.
- Knowledge of the DBMS configuration parameters that are in place.
- Data definition language (DDL) skills to translate the physical design into actual database objects.

Armed with the correct information, you can create an effective and efficient database from a logical data model. The first step in transforming a logical data model into a physical model is to perform a simple translation from logical terms to physical objects. Of course, this simple transformation will not result in a complete and correct physical database design – it is simply the first step. The transformation consists of the following things:

- Transforming entities into tables
- Transforming attributes into columns
- Transforming domains into data types and constraints

There are many decisions that must be made during the transition from logical to physical. For example, each of the following must be addressed:

- The nullability of each column in each table
- For character columns, should fixed length or variable length be used
- Should the DBMS be used to assign values to sequences or identity columns?
- Implementing logical relationships by assigning referential constraints
- Building indexes on columns to improve query performance
- Choosing the type of index to create: b-tree, bit map, reverse key, hash, partitioning, etc.
- Deciding on the clustering sequence for the data
- Other physical aspects such as column ordering, buffer pool specification, data files, denormalization, and so on.

Structured Query Language

SQL is an ANSI standard computer language for accessing and manipulating databases. SQL is standardized, and the current version is referred to as SQL-92. Any SQL-compliant database should conform to the standards of SQL at the time. If not,

they should state which flavor of SQL (SQL-89 for example) so that you can quickly figure out what features are and are not available. The standardization of SQL makes it an excellent tool for use in Web site design. Most Web application development toolkits, most notably Allaire's Cold Fusion and Microsoft's Visual InterDev, rely on SQL or SQL-like statements to connect to and extract information from databases. A solid foundation in SQL makes hooking databases to Web sites all the simpler. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database. This tutorial will provide you with the instruction on the basics of each of these commands as well as allow you to put them to practice using the SQL Interpreter.

Benefits of Standard SQL:
Following are the major benefits of SQL:-

Reduced training cost

- Application portability
- Application longevity
- Reduced dependence on a single vendor
- Cross-system communication**SQL is used for any type of interaction with the database through DBMS. It can be used for creating tables, insertion in the table and deletion as we well and other operations also.**

MS SQL Server

The DBMS for our course is Microsoft’s SQL Server 2000 desktop edition. There are two main tools Query Analyzer and Enterprise Manager; both can be used. For SQL practice we will use Query Analyzer. So you must use this software for the SQL queries. So we will be using this software for our SQL queries.

Summary:

In this lecture we have studied the vertical partitioning, its importance and methods of applying. We have also studied replication and clustering issues. We then started with the basics of SQL and in the next lectures we will use SQL Server for the queries.

Exercise:

Critically examine the tables drawn for Examination system and see if there is a requirement of vertical partitioning and then carry out the process. Also install the SQL Server and acquaint yourself with this software.

Lecture No. 25

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill

Overview of Lecture

- Structured Query Language (SQL)

In the previous lecture we have studied the partitioning and replication of data. From this lecture onwards we will study different rules of SQL for writing different commands.

Rules of SQL Format

SQL, at its simplest, is a basic language that allows you to "talk" to a database and extract useful information. With SQL, you may read, write, and remove information from a database. SQL commands can be divided into two main sub languages. The Data Definition Language (DDL) contains the commands used to create and destroy databases and database objects. After the database structure is defined with DDL, database administrators and users can utilize the Data Manipulation Language to insert, retrieve and modify the data contained within it. Following are the rules for writing the commands in SQL:-

- Reserved words are written in capital like SELECT or INSERT.
- User-defined identifiers are written in lowercase
- Identifiers should be valid, which means that they can start with @,_ alphabets ,or with numbers. The maximum length can be of 256. The reserved words should not be used as identifiers.
- Those things in the command which are optional are kept in [ ]
- Curly braces means required items
- | means choices
- [,…..n] means n items separated by comma

Consider the following example:-

SELECT [ALL|DISTINCT]

{*|select_list}

FROM {table|view[,…n]}

Select * from std

Data Types in SQL Server

In Microsoft SQL Server™, each column, local variable, expression, and parameter has a related data type, which is an attribute that specifies the type of data (integer, character, money, and so on) that the object can hold. SQL Server supplies a set of system data types that define all of the types of data that can be used with SQL Server. The set of system-supplied data types is shown below:-

Integers:

- Biggint

  Integer (whole number) data from -2^{63} (-9,223,372,036,854,775,808) through $2^{63}-1$ (9,223,372,036,854,775,807).

- Int

  Integer (whole number) data from -2^{31} (-2,147,483,648) through $2^{31} - 1$ (2,147,483,647).

- Smallint

  Integer data from -2^{15} (-32,768) through $2^{15} - 1$ (32,767).

- Tinyint

  Integer data from 0 through 255.

bit

Integer data with either a 1 or 0 value.

Decimal and Numeric

- Decimal

  Fixed precision and scale numeric data from $-10^{38} +1$ through $10^{38} -1$.

- Numeric

  Functionally equivalent to decimal.

Text:

It handles the textual data. Following are the different data types.

- Char: By default 30 characters, max 8000

- Varchar: Variable length text, max 8000
- Text: Variable length automatically
- nchar, nvarchar, ntext

Money:

It is used to handle the monetary data

- Small money: 6 digits, 4 decimal
- Money: 15 digits, 4 decimal

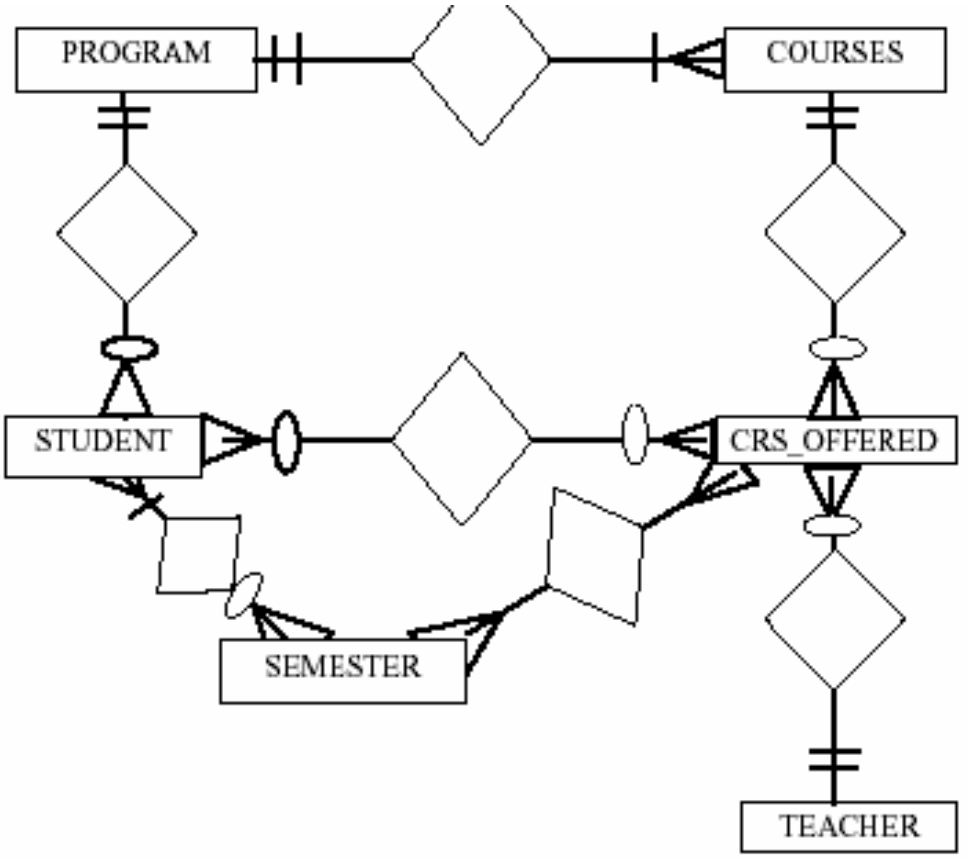
Floating point:

- Float
- Real

Date:

- Smalldatetime
- datetime

Examination System Database



We will now transfer this conceptual database design into relational database design as under:-

PROGRAM (prName, totSem, prCredits)

COURSE (crCode, crName, crCredits, prName)

SEMESTER (semName, stDate, endDate)

CROFRD (crCode, semName, facId)

FACULTY (facId, fName, fQual, fSal, rank)

STUDENT (stId, stName, stFName, stAdres,stPhone, prName, curSem, cgpa)

ENROLL (stId, crCode, semName, mTerm,sMrks, fMrks, totMrks, grade, gp)

SEM_RES (stId, semName, totCrs, totCrdts, totGP, gpa)

It is used to specify a database scheme as a set of definitions expressed in a DDL. DDL statements are compiled, resulting in a set of tables stored in a special file called

a data dictionary or data directory. The data directory contains metadata (data about data) the storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a data storage and definition language

Data Manipulation is retrieval, insertion, deletion and modification of information from the database. A DML is a language, which enables users to access and manipulate data. The goal is to provide efficient human interaction with the system. There are two types of DML.First is Procedural: in which the user specifies what data is needed and how to get it Second is Nonprocedural: in which the user only specifies what data is needed

The category of SQL statements that control access to the data and to the database. Examples are the GRANT and REVOKE statements.

Summary:

In today's lecture we have read about the basics of SQL. It is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database. In the end we have also seen the different types of SQL commands and their functions.

Exercise:

Practice the basic commands of SQL like SELECT, INSERT and CREATE.

Lecture No. 26

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill

Overview of Lecture

- Different Commands of SQL

In the previous lecture we have seen the database of an examination system. We had drawn the ER model and then the relational model, which was normalized. In this lecture we will now start with different commands of SQL.

Categories of SQL Commands

We have already read in our previous lecture that there are three different types of commands of SQL, which are DDL, DML and DCL. We will now study DDL.

DDL

It deals with the structure of database.The DDL (Data Definition Language) allows specification of not only a set of relations, but also the following information for each relation:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints.
- The set of indices for each relation.
- Security and authorization information.
- Physical storage structure on disk.

Following are the three different commands of DDL:-

Create

The first data management step in any database project is to create the database. This task can range from the elementary to the complicated, depending on your needs and the database management system you have chosen. Many modern systems (including Personal Oracle7) include graphical tools that enable you to completely build the database with the click of a mouse button. This timesaving feature is certainly helpful, but you should understand the SQL statements that execute in response to the mouse clicks. This command is used to create a new database table. The table is created in

the current default database. The name of the table must be unique to the database. The name must begin with a letter and can be followed by any combination of alphanumeric characters. The name is allowed to contain the underscore character (_). This command can be used to create permanent disk-based or temporary in-memory database tables. Data stored in a temporary table is lost when the server is shutdown. To create a temporary table the "**AS TEMP**" attribute must be specified. Note that querying against a temporary in-memory table is generally faster than querying against a disk-based table. This command is non-transactional. If no file size is given for a disk-based table, the table will be pre-allocated to 1MB. If no filegrowth is given, the default is 50%. It is used to create new tables, fields, views and indexes. It is used to create database. The format of statement is as under:
CREATE DATABASE db_name

For Example CREATE DATABASE EXAM. So now in this example database of exam has been created. Next step is to create tables. There are two approaches for creating the tables, which are:

- Through SQL Create command
- Through Enterprise Manager

Create table command is used to:

- Create a table
- Define attributes of the table with data types
- **Define different constraints on attributes, like primary and foreign keys, check constraint, not null, default value etc.**

The format of create table command is as under:

```
CREATE                                                    TABLE
  [     database_name.[     owner     ]     .     |     owner.     ]     table_name
  (               {               <               column_definition               >
    |             column_name             AS             omputed_column_expression
    | < table_constraint >
      }
      | [ { PRIMARY KEY | UNIQUE } [ ,...n ] ]
```

Let us now consider the CREATE statement used to create the Airport table definition for the Airline Database.

CREATE TABLE Airport

(airport char(4) not null,

name varchar(20),

checkin varchar(50),

resvtns varchar(12),

flightinfo varchar(12));

Table Name.(Airport)

The name chosen for a table must be a valid name for the DBMS.

Column Names. (Airport, Name, ..., FlightInfo)

The names chosen for the columns of a table must also be a valid name for the DBMS.

Data Types

Each column must be allocated an appropriate data type. In addition, key columns, i.e. columns used to uniquely identify individual rows of a given table, may be specified to be NOT NULL. The DBMS will then ensure that columns specified as NOT NULL always contain a value.

The column definition is explained as under:

< column_definition > ::= { *column_name data_type* }
[DEFAULT *constant_expression*]
[< column_constraint >] [*...n*]

The column constraint is explained as under:

< column_constraint > ::= [CONSTRAINT constraint_name]
{ [NULL | NOT NULL]
| [{ PRIMARY KEY | UNIQUE }
]
| [[FOREIGN KEY]
REFERENCES ref_table [**(** ref_column **)**]

[ON DELETE { CASCADE | NO ACTION }]
[ON UPDATE { CASCADE | NO ACTION }]
]

| CHECK**(** logical_expression **)**

}

)

We will now see some examples of CREATE command. This is a very simple command for creating a table.

```
CREATE TABLE Program (
        prName char(4),
        totSem tinyint,
        prCredits smallint)
```

If this command is to written in SQL Server, it will be written in Query Analyzer. We will now see an example in which has more attributes comparatively along with different data types:

```
CREATE TABLE Student
      (stId char(5),
       stName  char(25),
       stFName char(25),
       stAdres text,
       stPhone char(10),
        prName char(4)
       curSem  smallint,
       cgpa real)
```

In this example there are more attributes and different data types are also there. We will now see an example of creating a table with few constraints:

```
CREATE TABLE Student (
      stId  char(5)  constraint  ST_PK  primary  key   constraint ST_CK check (stId
      like‘S[0-          9][0-9][0-9][0-9]'),
       stName  char(25) not null,
       stFName char(25),
       stAdres text,
       stPhone char(10),
        prName char(4),
        curSem  smallint default 1,
         cgpa real)
```

Every constraint should be given a meaningful name as it can be referred later by its name. The check constraint checks the values for any particular attribute. In this way different types of constraints can be enforced in any table by CREATE command.

Summary

Designing a database properly is extremely important for the success of any application. In today's lecture we have seen the CREATE command of SQL. How different constraints are applied on this command with the help of different examples. This is an important command and must be practiced as it is used to create database and different tables. So create command is part of DDL.

Exercise:

Create a database of Exam System and create table of student with different constraints in SQL Server.

Lecture No. 27

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
"Teach Yourself SQL in 21 Days", Second Edition Que Series.

Overview of Lecture

Data Manipulation Language

In the previous lecture we were studying DDL in which we studied the CREATE command along with different examples. We also saw different constraints of create command. In this lecture we will study the ALTER and other SQL commands with examples.

Alter Table Statement

The purpose of ALTER statement is to make changes in the definition of a table already created through Create statement. It can add, and drop the attributes or constraints, activate or deactivate constraints. It modifies the design of an existing table. The format of this command is as under:

Syntax

ALTER TABLE table {

ADD [COLUMN] column type [(size)] [DEFAULT default] |

ALTER [COLUMN] column type [(size)] [DEFAULT default] |

ALTER [COLUMN] column SET DEFAULT default |

DROP [COLUMN] column |

RENAME [COLUMN] column TO columnNew

}

The ALTER TABLE statement has these parts:

Part	Description
Table	The name of the table to be altered.

Column	The name of the column to be altered or added to or deleted from table.
ColumnNew	The new name of the altered column
Type	The data type of column.
Size	The size of the altered column in characters or bytes for text or binary columns.
Default	An expression defining the new default value of the altered column. Can contain literal values, and functions of these values

Using the ALTER TABLE statement, we can alter an existing table in several ways. We can:

- Use ADD COLUMN to add a new column to the table. Specify the name, data type, an optional size, and an optional default value of the column.
- Use ALTER COLUMN to alter type, size or default value of an existing column.
- Use DROP COLUMN to delete a column. Specify only the name of the column.
- Use RENAME COLUMN to rename an existing column. We cannot add, delete or modify more than one column at a time.We will now see an example of alter command

```
ALTER TABLE Student
   add constraint fk_st_pr
   foreign key  (prName) references
                    Program (prName)
```

This is a simple example, in which we have incorporated a constraint and the names are meaningful, so that if in the future we have to refer them, we can do so. We will now see an example of removing or changing attribute.

```
ALTER TABLE student
     ALTER COLUMN stFName char (20)
ALTER TABLE student
    Drop column curSem
ALTER TABLE student
   Drop constraint ck_st_pr
```

Now in these examples either an attribute is deleted or altered by using the keywords of Drop and Alter. We will now see an example in which few or all rows will be removed, or whole table is required to be removed. The TRUNCATE is used to delete all the rows of any table but rows would exist. The DELETE is used to delete one or many records. If we want to remove all records we must use TRUNCATE. Next is the DROP table command, which is used to drop the complete table from the database.

```
TRUNCATE TABLE table_name
Truncate table class
```

Delete can also be used
DROP TABLE table_name

Data Manipulation Language
The non-procedural nature of SQL is one of the principle characteristics of all 4GLs - Fourth Generation Languages - and contrasts with 3GLs (eg, C, Pascal, Modula-2, COBOL, etc) in which the user has to give particular attention to how data is to be accessed in terms of storage method, primary/secondary indices, end-of-file conditions, error conditions (eg, Record NOT Found), and so on. The Data Manipulation Language (DML) is used to retrieve, insert and modify database information. Data Manipulation is retrieval, insertion, deletion and modification of information from the database SQL is a non-procedural language that is, it allows the user to concentrate on specifying what data is required rather than concentrating on the how to get it. There are two types of DML.First is procedural in which: the user specifies what data is needed and how to get it. Second is nonprocedural in which the user only specifies what data is needed. The DML component of SQL comprises of following basic statements:
Insert To add new rows to tables.
Select To retrieve rows from tables
Update To modify the rows of tables

Insert
The INSERT command in SQL is used to add records to an existing table. We will now see the format of insert command as under:

```
INSERT [INTO] table
    {[ ( column_list ) ]
     { VALUES
        ( { DEFAULT | NULL | expression } [ ,...n] )
      }
  }
  | DEFAULT VALUES
```

The basic format of the INSERT...VALUES statement adds a record to a table using the columns you give it and the corresponding values you instruct it to add. You must follow three rules when inserting data into a table with the INSERT...VALUES statement:
The values used must be the same data type as the fields they are being added to.
The data's size must be within the column's size. For instance, you cannot add an 80-character string to a 40-character column.
The data's location in the VALUES list must correspond to the location in the column list of the column it is being added to. (That is, the first value must be entered into the first column, the second value into the second column, and so on.)
The rules mentioned above must be followed. We will see the examples of the insert statement in the coming lectures.
Summary
SQL provides three statements that can be used to manipulate data within a database. The INSERT statement has two variations. The INSERT...VALUES statement inserts a set of values into one record. The INSERT...SELECT statement is used in combination with a SELECT statement to insert multiple records into a table based on the contents of one or more tables. The SELECT statement can join multiple tables, and the results of this

join can be added to another table. The UPDATE statement changes the values of one or more columns based on some condition. This updated value can also be the result of an expression or calculation.

The DELETE statement is the simplest of the three statements. It deletes all rows from a table based on the result of an optional WHERE clause. If the WHERE clause is omitted, all records from the table are deleted. Modern database systems supply various tools for data manipulation. Some of these tools enable developers to import or export data from foreign sources. This feature is particularly useful when a database is upsized or downsized to a different system. Microsoft Access, Microsoft and Sybase SQL Server, and Personal Oracle7 include many options that support the migration of data between systems.

Exercise:

Try inserting values with incorrect data types into a table. Note the errors and then insert values with correct data types into the same table.

Lecture No. 28

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
"Teach Yourself SQL in 21 Days", Second Edition Que Series.

In the previous lecture we started the data manipulation language, in which we were discussing the Insert statement, which is used to insert data in an existing table. In today's lecture we will first see an example of Insert statement and then discuss the other SQL Commands.

The INSERT statement allows you to insert a single record or multiple records into a table. It has two formats:

INSERT INTO table-1 [(column-list)] VALUES (value-list)

And,

INSERT INTO table-1 [(column-list)] (query-specification)

The first form inserts a single row into table-1 and explicitly specifies the column values for the row. The second form uses the result of query-specification to insert one or more rows into table-1. The result rows from the query are the rows added to the insert table. Both forms have an optional column-list specification. Only the columns listed will be assigned values. Unlisted columns are set to null, so unlisted columns must allow nulls. The values from the VALUES Clause (first form) or the columns from the query-specification rows (second form) are assigned to the corresponding column in column-list in order. If the optional column-list is missing, the default column list is substituted. The default column list contains all columns in table-1 in the order they were declared in CREATE TABLE.

The VALUES Clause in the INSERT Statement provides a set of values to place in the columns of a new row. It has the following general format:

VALUES (value-1 [, value-2]...)

Value-1 and value-2 are Literal Values or Scalar Expressions involving literals. They can also specify NULL. The values list in the VALUES clause must match the

explicit or implicit column list for INSERT in degree (number of items). They must also match the data type of corresponding column or be convertible to that data type.

We will now see an example of INSERT statement for that we have the table of COURSE with following attributes: -

COURSE (crCode, crName, crCredits, prName)

The INSERT statement is as under:

INSERT INTO course VALUES ('CS-211', 'Operating Systems', 4, 'MCS')

This is a simple INSERT statement; we have not used the attribute names because we want to enter values for all the attributes. So here it is important to enter the values according to the attributes and their data types. We will now see an other example of insert statement:

INSERT INTO course (crCode, crName) VALUES ('CS-316', Database Systems')

In this example we want to enter the values of only two attributes, so it is important that other two attributes should not be NOT NULL. So in this example we have entered values of only two particular attributes. We will now see another example of INSERT statement as under:

INSERT INTO course ('MG-103', 'Intro to Management', NULL, NULL)

In this example we have just entered the values of first two attributes and rest two are NULL. So here we have not given the attribute names and just placed NULL in those values.

Select Statement

Select statement is the most widely used SQL Command in Data Manipulation Language. It is not only used to select rows but also the columns. The SQL SELECT statement queries data from tables in the database. The statement begins with the SELECT keyword. The basic SELECT statement has 3 clauses:

- SELECT
- FROM
- WHERE

The SELECT clause specifies the table columns that are retrieved. The FROM clause specifies the tables accessed. The WHERE clause specifies which table rows are used. The WHERE clause is optional; if missing, all table rows are used. The SELECT clause is mandatory. It specifies a list of columns to be retrieved from the tables in the FROM clause. The FROM clause always follows the SELECT clause. It lists the tables accessed by the query. The WHERE clause is optional. When specified, it always follows the FROM clause. The WHERE clause filters rows from the FROM clause tables. Omitting the WHERE clause specifies that all rows are used. The syntax for the SELECT statement is:

SELECT {*|col_name[,….n]} FROM table_name

This is the simplest form of SELECT command. In case of * all the attributes of any table would be available. If we do not mention the * then we can give the names of particular attribute names. Next is the name of the table from where data is required. We will now see different examples of SELECT statement using the following table:
STUDENT

stId	**stName**	**prName**	**cgpa**
S1020	Sohail Dar	MCS	2.8
S1038	Shoaib Ali	BCS	2.78
S1015	Tahira Ejaz	MCS	3.2
S1034	Sadia Zia	BIT	
S1018	Arif Zia	BIT	3.0

So the first query is
Q: Get the data about studentsSELECT * FROM students
The output of this query is as under:

	stId	**stName**	**prName**	**cgpa**
1	S1020	Sohail Dar	MCS	2.8
2	S1038	Shoaib Ali	BCS	2.78
3	S1015	Tahira Ejaz	MCS	3.2
4	S1034	Sadia Zia	BIT	
5	S1018	Arif Zia	BIT	3.0

We will now see another query, in which certain specific data is required form the table: The query is as under:
Q: Give the name of the students with the program nameThe SQL Command for the query is as under:

SELECT stName, prName
FROM student
The output for the command is as under:

	stName	**prName**
1	Sohail Dar	MCS

2	Shoaib Ali	BCS
3	Tahira Ejaz	MCS
4	Sadia Zia	BIT
5	Arif Zia	BIT

Attribute Allias

SELECT {*|col_name [[AS] alias] [, …n]} FROM tab_name

Now in this case if all the attributes are to be selected by * then we cannot give the name of attributes. The AS is also optional here then we can write the name of attribute what we want. We will now see an example.

SELECT stName as 'Student Name', prName 'Program' FROM Student

The output of this query will be as under:

	Student Name	**Program**
1	Sohail Dar	MCS
2	Shoaib Ali	BCS
3	Tahira Ejaz	MCS
4	Sadia Zia	BIT
5	Arif Zia	BIT

In the column list we can also give the expression; value of the expression is computed and displayed. This is basically used where some arithmetic operation is performed, in which that operation is performed on each row and then that result is displayed as an output. We will now see it with an example:
Q Display the total sessional marks of each student obtained in each subject

The SQL Command for the query will be as under:

Select stId, crCode, mTerm + sMrks 'Total out of 50' from enroll

The DISTINCT keyword is used to return only distinct (different) values. The SELECT statement returns information from table columns. But what if we only want to select distinct elements With SQL, all we need to do is to add a DISTINCT keyword to the SELECT statement. The format is as under:

SELECT DISTINCT column_name(s)

FROM table_name

We will now see it with an example

Q Get the program names in which students are enrolled

The SQL Command for this query is as under:

SELECT DISTINCT prName FROM Student

	programs
1	BCS
2	BIT
3	MCS
4	MBA

The "WHERE" clause is optional. When specified, it always follows the FROM clause. The "WHERE" clause filters rows from "FROM" clause tables. Omitting the WHERE clause specifies that all rows are used. Following the WHERE keyword is a logical expression, also known as a predicate. The predicate evaluates to a SQL logical value -- true, false or unknown. The most basic predicate is a comparison:

Color = 'Red'

This predicate returns:

- True -- If the color column contains the string value -- 'Red',
- False -- If the color column contains another string value (not 'Red'), or
- Unknown -- If the color column contains null.

Generally, a comparison expression compares the contents of a table column to a literal, as above. A comparison expression may also compare two columns to each other. Table joins use this type of comparison.

In today's we have studied the SELECT statement with different examples. The keywords SELECT and FROM enable the query to retrieve data. You can make a broad statement and include all tables with a SELECT * statement or you can rearrange or retrieve specific tables. The keyword DISTINCT limits the output so that you do not see duplicate values in a column. In the coming lecture we will see further SQL Commands of Data Manipulation Language.

Lecture No. 29

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
"Teach Yourself SQL in 21 Days", Second Edition Que Series.

Overview of Lecture

Data Manipulation Language

In the previous lecture we have studied the SELECT statement, which is the most widely used SQL statement. In this lecture we will study the WHERE clause. This is used to select certain specific rows.

The WHERE clause allows you to filter the results from an SQL statement - select, insert, update, or delete statement. The rows which satisfy the condition in the where clause are selected. The format of WHERE clause is as under:

SELECT [ALL|DISTINCT]

{*|column_list [alias][,…..n]} FROM table_name

[WHERE <search_condition>]

Here WHERE is given in square brackets, which means it is optional. We will see the search condition as under:

Search Condition

{ [NOT] < predicate > | (< search_condition >) }

[{ AND | OR } [NOT] { < predicate > |

(< search_condition >) }]

} [,...n]

< predicate > ::=

{ expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }

```
                    expression
    | string_expression [ NOT ] LIKE string_expression
    | expression [ NOT ] BETWEEN expression AND
          expression
    | expression IS [ NOT ] NULL
    | expression [ NOT ] IN ( subquery | expression [ ,...n ] )
    | expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
      { ALL | SOME | ANY} ( subquery )
    | EXISTS ( subquery )
}
```

In this format where clause is used in expressions using different comparison operators. Those rows, which fulfill the condition, are selected in the output.

```
SELECT  *
FROM supplier
WHERE supplier_name = 'IBM';
```

In this first example, we have used the WHERE clause to filter our results from the supplier table. The SQL statement above would return all rows from the supplier table where the supplier_name is IBM. Because the * is used in the select, all fields from the supplier table would appear in the result set. We will now see another example of where clause.

```
SELECT supplier_id
FROM supplier
WHERE supplier_name = 'IBM'
or supplier_city = 'Karachi';
```

We can define a WHERE clause with multiple conditions. This SQL statement would return all supplier_id values where the supplier_name is IBM or the supplier_city is Karachi..

```
SELECT supplier.suppler_name, orders.order_id
FROM supplier, orders
WHERE supplier.supplier_id = orders.supplier_id
and supplier.supplier_city = 'Karachi';
```

We can also use the WHERE clause to join multiple tables together in a single SQL statement. This SQL statement would return all supplier names and order_ids where there is a matching record in the supplier and orders tables based on supplier_id, and where the supplier_city is Karachi.

We will now see a query in which those courses, which are part of MCS, are to be displayed

Q: Display all courses of the MCS program

Select crCode, crName, prName from course

where prName = 'MCS

Now in this query whole table would be checked row by row and where program name would be MCS would be selected and displayed.'

Q List the course names offered to programs other than MCS

SELECT crCode, crName, prName

FROM course

WHERE not (prName = 'MCS')

Now in this query again all the rows would be checked and those courses would be selected and displayed which are not for MCS. So it reverses the output.

The BETWEEN condition allows you to retrieve values within a specific range.

The syntax for the BETWEEN condition is:

```
SELECT columns
FROM tables
WHERE column1 between value1 and value2;
```

This SQL statement will return the records where column1 is within the range of value1 and value2 (inclusive). The BETWEEN function can be used in any valid SQL statement - select, insert, update, or delete. We will now see few examples of this operator.

```
SELECT *
FROM suppliers
WHERE supplier_id between 10 AND 50;
```

This would return all rows where the supplier_id is between 10 and 50.

The BETWEEN function can also be combined with the NOT operator.

For example,

```
SELECT *
FROM suppliers
WHERE supplier_id not between 10 and 50;
```

The IN function helps reduce the need to use multiple OR conditions. It is sued to check in a list of values. The syntax for the IN function is:

```
SELECT columns
FROM tables
WHERE column1 in (value1, value2,.... value_n);
```

This SQL statement will return the records where column1 is value1, value2... or value_n. The IN function can be used in any valid SQL statement - select, insert, update, or delete. We will now see an example of IN operator.

SELECT crName, prName
From course
Where prName in (‘MCS’, ‘BCS’)
It is equal to the following SQL statement
SELECT crName, prName
From course
Where (prName = ‘MCS’) OR (prName = ‘BCS’)
Now in these two queries all the rows will be checked for MCS and BCS one by one so OR can be replaced by IN operator.

The LIKE operator allows you to use wildcards in the where clause of an SQL statement. This allows you to perform pattern matching. The LIKE condition can be used in any valid SQL statement - select, insert, update, or delete.

The patterns that you can choose from are:

% Allows you to match any string of any length (including zero length)

_ Allows you to match on a single character

We will now see an example of LIKE operator

Q: Display the names and credits of CS programs

```
SELECT crName, crCrdts, prName FROM course
WHERE prName like '%CS'
```

The ORDER BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements.

The syntax for the ORDER BY clause is:

```
SELECT columns
FROM tables
WHERE predicates
ORDER BY column ASC/DESC;
```

The ORDER BY clause sorts the result set based on the columns specified. If the ASC or DESC value is omitted, the system assumed ascending order.

ASC indicates ascending order. (Default)

DESC indicates descending order.

We will see the example of ORDER BY clause in our next lecture.

In today's lecture we have discussed different operators and use of WHERE clause which is the most widely used in SQL Commands. These different operators are used according to requirements of users. We will study rest of the SQL Commands in our coming lectures.

Lecture No. 30

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill

Overview of Lecture

Data Manipulation Language
Functions in SQL

In the previous lecture we have discussed different operators of SQL, which are used in different commands. By the end of previous lecture we were discussing ORDER BY clause, which is basically used to bring the output in ascending or descending order. In this lecture we will see some examples of this clause.

ORDER BY Clause

The ORDER BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements. The ORDER BY clause sorts the result set based on the columns specified. If the ASC or DESC value is omitted, the system assumed ascending order. We will now see few examples of this clause

```
SELECT supplier_city
FROM supplier
WHERE supplier_name = 'IBM'
ORDER BY supplier_city;
```

This would return all records sorted by the supplier_city field in ascending order.

```
SELECT supplier_city
FROM supplier
```

```
WHERE supplier_name = 'IBM'
ORDER BY supplier_city DESC;
```

This would return all records sorted by the supplier_city field in descending order.

Functions in SQL

A function is a special type of command. Infact, functions are one-word command that return a single value. The value of a function can be determined by input parameters, as with a function that averages a list of database values. But many functions do not use any type of input parameter, such as the function that returns the current system time, CURRENT_TIME. There are normally two types of functions. First is Built in, which are provided by any specific tool or language. Second is user defined, which are defined by the user. The SQL supports a number of useful functions.. In addition, each database vendor maintains a long list of their own internal functions that are outside of the scope of the SQL standard.

Categories of Functions:

These categories of functions are specific to SQL Server. Depending on the arguments and the return value, functions are categorized as under:

- Mathematical (ABS, ROUND, SIN, SQRT)
- String (LOWER, UPPER, SUBSTRING, LEN)
- Date (DATEDIFF, DATEPART, GETDATE ())
- System (USER, DATALENGTH, HOST_NAME)
- Conversion (CAST, CONVERT)

We will now see an example using above-mentioned functions:

```
SELECT upper (stName), lower (stFName), stAdres, len(convert(char, stAdres)),
  FROM student
```

In this example student name will be displayed in upper case whereas father name will be displayed in lower case. The third function is of getting the length of student address. It has got nesting of functions, first address is converted into character and then its length will be displayed.

Aggregate Functions

These functions operate on a set of rows and return a single value. If used among many other expressions in the item list of a SELECT statement, the SELECT must

have a GROUP BY clause. No GROUP BY clause is required if the aggregate function is the only value retrieved by the SELECT statement. Following are some of the aggregate functions:

Function	Usage
AVG(expression)	Computes average value of a column by the expression
COUNT(expression)	Counts the rows defined by the expression
COUNT(*)	Counts all rows in the specified table or view
MIN(expression)	Finds the minimum value in a column by the expression
MAX(expression)	Finds the maximum value in a column by the expression
SUM(expression)	Computes the sum of column values by the expression

```
SELECT avg(cgpa) as 'Average CGPA', max(cgpa) as 'Maximum CGPA'
 from student
```

GROUP BY Clause

The GROUP BY clause can be used in a SELECT statement to collect data across multiple records and group the results by one or more columns. It is added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it is impossible to find the sum for each individual group of column values.

The syntax for the GROUP BY clause is:

```
SELECT column1, column2, ... column_n, aggregate_function
(expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n;
```

Aggregate function can be a function such as SUM, COUNT, MIN or MAX

Example using the SUM function
For example, the SUM function can be used to return the name of the department and the total sales (in the associated department).

```
SELECT department, SUM (sales) as "Total sales"
FROM order_details
GROUP BY department;
```

In this example we have listed one column in the SELECT statement that is not encapsulated in the SUM function, so we have used a GROUP BY clause. The department field must, therefore, be listed in the GROUP BY section.

Example using the COUNT function
We can also use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over Rs 25,000 / year.

```
SELECT department, COUNT (*) as "Number of employees"
FROM employees
WHERE salary > 25000
GROUP BY department;
```

HAVING Clause

The HAVING clause is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records that a GROUP BY returns. At times we want to limit the output based on the corresponding sum (or any other aggregate functions). For example, we might want to see only the stores with sales over Rs 1,500. Instead of using the WHERE clause in the SQL statement, though, we need to use the HAVING clause, which is reserved for aggregate functions. The HAVING clause is typically placed near the end of the SQL statement, and a SQL statement with the HAVING clause may or may not include the GROUP BY clause. The syntax for the HAVING clause is:

```
SELECT column1, column2, ... column_n, aggregate_function
(expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n
HAVING condition1 ... condition_n;
```

Aggregate function can be a function such as SUM, MIN or MAX.

We will now see few examples of HAVING Clause.

Example using the SUM function

We can use the SUM function to return the name of the department and the total sales (in the associated department). The HAVING clause will filter the results so that only departments with sales greater than Rs 1000 will be returned.

SELECT department, SUM (sales) as "Total sales"
FROM order_details
GROUP BY department
HAVING SUM (sales) > 1000;

Example using the COUNT function

For example, you could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over $25,000 / year. The HAVING clause will filter the results so that only departments with at least 25 employees will be returned.

SELECT department, COUNT (*) as "Number of employees"
FROM employees
WHERE salary > 25000
GROUP BY department
HAVING COUNT (*) > 10;

Accessing Multiple Tables:

Until now we have been accessing data through one table only. But there can be occasions where we have to access the data from different tables. So depending upon different requirements data can be accessed from different tables. Referential integrity constraint plays an important role in gathering data from multiple tables. Following are the methods of accessing data from different tables:

Cartesian Product

- Inner join
- Outer Join
- Full outer join
- Semi Join
- Natural JoinWe will now discuss them one by one.

Cartesian product:

A Cartesian join gives a Cartesian product. A Cartesian join is when you join every row of one table to every row of another table. You can also get one by joining every row of a table to every row of itself. No specific command is used just Select is used to join two tables. Simply the names of the tables involved are given and Cartesian product is produced. It produces m x n rows in the resulting table. We will now see few examples of Cartesian product.

Select * from program, course

Now in this example all the attributes of program and course are selected and the total number of rows would be number of rows of program x number of rows of course.In Cartesian product certain columns can be selected, same column name needs to be

qualified. Similarly it can be applied to more than one table, and even can be applied on the same table .For Example

SELECT * from Student, class, program

Summary

In today's lecture we have seen certain important functions of SQL, which are more specific to SQL Server. We studied some mathematical, string and conversion functions, which are used in SQL Commands. We also studied Aggregate functions, which are applied on a entire table or a set of rows and return one value. We also studied Group By clause which is used in conjunction with aggregate functions. In the end we saw how to extract data from different tables and in that we studied Cartesian product. We will see rest of the methods in our coming lectures.

Lecture No. 31

Reading Material

Raghu Ramakrishnan, Johannes Gehkre, 'Database Management Systems', Second edition	Chapter 17

Overview of Lecture

- Types of Joins
- Relational Calculus
- Normalization

In the previous lecture we studied that rows from two tables can be merged with each other using the Cartesian product. In real life, we very rarely find a situation when two tables need to be merged the way Cartesian product, that is, every row of one table is merged with every row of the other table. The form of merging that is useful and used most often is 'join'. In the following, we are going to discuss different forms of join.

Inner Join

Only those rows from two tables are joined that have same value in the common attribute. For example, if we have two tables R and S with schemes

R (a, b, c, d) and S (f, r, h, a), then we have 'a' as common attribute between these twit tables. The inner join between these two tables can be performed on the basis of 'a' which is the common attribute between the two. The common attributes are not required to have the same name in both tables, however, they must have the same domain in both tables. The attributes in both tables are generally tied in a primary-foreign key relationship but that also is not required. Consider the following two tables:

	crCode	crName	crCrdts	prName
1	CS-105	Intro to Programmin...	4	BCS
2	CS-216	Data Structures	4	BCS
3	CS-316	Database Systems	4	BCS
4	CS-504	Analysis of Algorit...	4	MCS
5	CS-511	Operating System	4	MCS
6	CS-516	Data Structures and...	4	MCS
7	CS-616	Intro to Database S...	3	MCS
8	MG-103	Intro to Management...	NULL	NULL
9	MG-105	Intro to Accounting...	3	BBA
10	MG-314	Money & Capital Mar...	3	BIT
11	MG-505	Intro to Accounting...	3	MBA
12	MT-305	Linear Algebra	3	MCS

	prName	totSem	prCredits
1	BBA	8	130
2	BCS	8	134
3	BIT	8	132
4	MBA	4	64
5	MCS	4	64

Fig. 1: COURSE and PROGRAM tables with common attribute prName

The figure shows two tables, COURSE and PROGRAM. The COURSE.prName and PROGRAM. prName are the common attributes between the two tables; incidentally the attributes have the same names and definitely the same domains. If we apply inner join on these tables, the rows from both tables will be merged based on the values of common attribute, that is, the prName. Like, row one of COURSE has the value 'BCS' in attribute prName. On the other hand, row number 2 in PROGRAM table has the value 'BCS'. So these two rows will merge and form one row of the resultant table of the inner join operation. As has been said before, the participating tables of inner join are generally tied in a primary-foreign key link, so the common attribute is PK in one of the tables. It means the table in which the common attribute is FK, the rows from this table will not be merged with more that one row from the other table. Like in the above example, each row from COURSE table will find exactly one match in PROGRAM table, since the prName is the PK in PROGRAM table.

The inner join can be implemented using different techniques. One possibility is that we may find 'inner join' operation as such, like:

- **SELECT * FROM course INNER JOIN program ON course.prName = program.prName**

or

- **Select * FROM Course c INNER JOIN program p ON c.prName = p.prName**

The output after applying inner join on tables of figure 1 will be as follows:

	crCode	crName	crCrdts	prName	prName	totSem	prCredits
1	CS-105	Intro to Programming	4	BCS	BCS	8	134
2	CS-216	Data Structures	4	BCS	BCS	8	134
3	CS-316	Database Systems	4	BCS	BCS	8	134
4	CS-504	Analysis of Algorithm	4	MCS	MCS	4	64
5	CS-511	Operating System	4	MCS	MCS	4	64
6	CS-516	Data Structures and Algos	4	MCS	MCS	4	64
7	CS-616	Intro to Database Systems	3	MCS	MCS	4	64
8	MG-105	Intro to Accounting	3	BBA	BBA	8	130
9	MG-314	Money & Capital Mark	3	BIT	BIT	8	132
10	MG-505	Intro to Accounting	3	MBA	MBA	4	64
11	MT-305	Linear Algebra	3	MCS	MCS	4	64

Fig. 2: Output of inner join on tables of figure 1

As can be seen in the figure, the common attribute appears twice in the output of inner join; that is, from both the tables. Another possible approach to implement inner join can be as follows:

SELECT * FROM course, program WHERE course.prName = program.prName

The output of this statement will be exactly the same as is given in figure 2.

Outer Join

SQL supports some interesting variants of the join operation that rely on null values, called outer joins. Consider the two tables COURSE and PROGRAM given in figure 1 and their inner join given in figure 2. Tuples of COURSE that do not match some row in PROGRAM according to the inner join condition (COURSE.prName = PROGRAM.prName) do not appear in the result. In an outer join, on the other hand, COURSE rows without a matching PROGRAM row appear exactly once in the result, with the result columns inherited from PROGRAM assigned null values.

In fact, there are several variants of the outer join idea. In a right outer join, COURSE rows without a matching PROGRAM row appear in the result, but not vice versa. In a left outer join, PROGRAM rows without a matching COURSE row appear in the result, but not vice versa. In a full outer join, both COURSE and PROGRAM rows without a match appear in the result. (Of course, rows with a match always appear in the result, for all these variants, just like the usual joins or inner joins).

SQL-92 allows the desired type of join to be specified in the FROM clause. For example,

- **Select * from COURSE c RIGHT OUTER JOIN**

PROGRAM p on c.prName = p.prName

	crCode	crName	crCrdts	prName	prName	totSem	prCredits
1	CS-105	Intro to Programming	4	BCS	BCS	8	134
2	CS-216	Data Structures	4	BCS	BCS	8	134
3	CS-316	Database Systems	4	BCS	BCS	8	134
4	CS-504	Analysis of Algorithm	4	MCS	MCS	4	64
5	CS-511	Operating System	4	MCS	MCS	4	64
6	CS-516	Data Structures and Algos	4	MCS	MCS	4	64
7	CS-616	Intro to Database Systems	3	MCS	MCS	4	64
8	MG-103	Intro to Management	NULL	NULL	NULL	NULL	NULL
9	MG-105	Intro to Accounting	3	BBA	BBA	8	130
10	MG-314	Money & Capital Mark	3	BIT	BIT	8	132
11	MG-505	Intro to Accounting	3	MBA	MBA	4	64
12	MT-305	Linear Algebra	3	MCS	MCS	4	64

Fig. 3: Right outer join of the tables in figure 1

In figure 3 above, the row number 8 is the non matching row of COURSE that contains nulls in the attributes corresponding to PROGRAM table, rest of the rows are the same as in inner join of figure 2.

- **Select * from COURSE c LEFT OUTER JOIN**

PROGRAM p on c.prName = p.prName

	crCode	crName	crCrdts	prName	prName	totSem	prCredits
1	MG-105	Intro to Accounting...	3	BBA	BBA	8	130
2	CS-105	Intro to Programmin...	4	BCS	BCS	8	134
3	CS-216	Data Structures	4	BCS	BCS	8	134
4	CS-316	Database Systems	4	BCS	BCS	8	134
5	MG-314	Money & Capital Mar...	3	BIT	BIT	8	132
6	MG-505	Intro to Accounting...	3	MBA	MBA	4	64
7	CS-504	Analysis of Algorit...	4	MCS	MCS	4	64
8	CS-511	Operating System	4	MCS	MCS	4	64
9	CS-516	Data Structures and...	4	MCS	MCS	4	64
10	CS-616	Intro to Database S...	3	MCS	MCS	4	64
11	MT-305	Linear Algebra	3	MCS	MCS	4	64
12	NULL	NULL	NULL	NULL	MIT	4	62

Fig. 4: Left outer join of the tables in figure 1

In figure 4 above, the row number 12 is the non matching row of PROGRAM that contains nulls in the attributes corresponding to COURSE table, rest of the rows are the same as in inner join of figure 2.

- **Select * from COURSE c FULL OUTER JOIN**

  PROGRAM p on c.prName = p.prName

	prName	totSem	prCredits	crCode	crName	crCrdts	prName
1	NULL	NULL	NULL	MG-103	Intro to Management	NULL	NULL
2	BBA	8	130	MG-105	Intro to Accounting	3	BBA
3	BCS	8	134	CS-105	Intro to Programming	4	BCS
4	BCS	8	134	CS-216	Data Structures	4	BCS
5	BCS	8	134	CS-316	Database Systems	4	BCS
6	BIT	8	132	MG-314	Money & Capital Mark	3	BIT
7	MBA	4	64	MG-505	Intro to Accounting	3	MBA
8	MCS	4	64	MT-305	Linear Algebra	3	MCS
9	MCS	4	64	CS-504	Analysis of Algorithm	4	MCS
10	MCS	4	64	CS-616	Intro to Database Systems	3	MCS
11	MCS	4	64	CS-511	Operating System	4	MCS
12	MCS	4	64	CS-516	Data Structures and Algos	4	MCS
13	MIT	4	62	NULL	NULL	NULL	NULL

Fig. 5: Full outer join of the tables in figure 1

In figure 5 above, the row number 1 and 13 are the non matching rows from both tables, rest of the rows are the same as in inner join of figure 2.

Semi Join

Another form of join that involves two operations. First inner join is performed on the participating tables and then resulting table is projected on the attributes of one table. The advantage of this operation is that we can know the particular rows of one table that are involved in inner join. For example, through semi join of COURSE and PROGRAM tables we will get the rows of COURSE that have matching rows in PROGRAM, or in other words, the courses that are part of any program. Same can be performed other way round. SQL does not provide any operator as such, but can be implemented by select and inner join operations, for example.

- **SELECT distinct p.prName, totsem, prCredits FROM program p inner JOIN course c ON p.prName = c.prName**

	prName	totsem	prCredits
1	BBA	8	130
2	BCS	8	134
3	BIT	8	132
4	MBA	4	64
5	MCS	4	64

Fig. 6: Semi join of tables in figure 1

Self Join

In self join a table is joined with itself. This operation is used when a table contains the reference of itself through PK, that is, the PK and the FK are both contained in the same table supported by the referential integrity constraint. For example, consider STUDENT table having an attribute 'cr' storing the id of the student who is the class representative of a particular class. The example table is shown in figure 7, where a CR has been specified for the MCS class, rest of the class students contain a null in the 'cr' attribute.

	stId	stname	prName	cr
1	S0123	Amjad	MCS	NULL
2	S1012	Amjad	MCS	S0123
3	S1015	Tahira Ejaz	MCS	S0123
4	S1018	Arif Zia	BIT	NULL
5	S1020	Suhai Dar	MCS	S0123
6	S1021	M. Ali	MBA	NULL
7	S1034	Sadia Zia	BIT	NULL

Fig. 7: Example STUDENT table

Applying self join on this table:

- **SELECTa.stId, a.stName, b.stId, b.stName FROM student a, student b**
  WHERE a.cr = b.stId

Since same table is involved two times in the join, we have to use the alias. The above statement displays the names of the students and of the CR.

	stId	stName	stId	stName
1	S1012	Amjad	S0123	Amjad
2	S1015	Tahira Ejaz	S0123	Amjad
3	S1020	Suhai Dar	S0123	Amjad

Fig. 8: Self join of STUDENT table of figure 7

Subquery

Subquery is also called nested query and is one of the most powerful features of SQL. A nested query is a query that has another query embedded within it; the embedded query is called a subquery. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause. Here we have discussed only subqueries that appear in the WHERE clause. The treatment of subqueries appearing elsewhere is quite similar. Examples of subqueries that appear in the FROM clause are discussed in following section.

Lets suppose we want to get the data of the student with the maximum cgpa, we cannot get them within a same query since to get the maximum cgpa we have to apply the group function and with group function we cannot list the individual attributes. So we use nested query here, the outer query displays the attributes with the condition on cgpa whereas the subquery finds the maximum cgpa as shown below:

- SELECT * from student where cgpa >
  (select max(cgpa) from student where prName = 'BCS‘)

	stId	stName	stFName	stAdres	stPhone	prName	curSem	cgpa
1	S0123	Amjad	Hussain	8-SD Lahore	234322	MCS	1	NULL
2	S1012	Amjad	Rehan	I8 Ibd	5456754	MCS	2	2.3
3	S1018	Arif Zia	Zia Khan	GM Rawalpindi	4356488	BIT	2	2.8
4	S1021	M. Ali	M. Saad	JT Lahore	544325	MBA	2	2.8
5	S1034	Sadia Zia	NULL	NULL	NULL	BIT	1	NULL
6	S1038	Shoaib Ali	Rahmat Ali	G-6 Islamabad	5343240	BCS	2	3.2
7	S1020	Suhai Dar	Nek Muhammad	I-8 Islamabad	5523240	MCS	2	3.2
8	S1015	Tahira Ejaz	Nek Muhammad	AJ Road	4323456	MCS	3	3.2

	stId	stName	stFName	stAdres	stPhone	prName	curSem	cgpa	cr
1	S1015	Tahira Ejaz	Nek Muhammad	AJ Road	4323456	MCS	3	3.25	S012

Fig. 9: STUDENT table and nested query applied on it

We have to take care of the operator being applied in case of subquery in the where clause. The type of operator depends on the result set being returned by the subquery. If the output expected from the subquery is a single value, as is the case in the above example, then we can use operators like =, <, >, etc. However, if the subquery returns multiple values then we can use operators like IN, LIKE etc. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. We can also use the NOT IN operator where required.

The subquery can be nested to any level, the queries are evaluated in the reverse order, and that is, the inner most is evaluated first, then the outer one and finally the outer most.

ACCESS CONTROL

SQL-92 supports access control through the GRANT and REVOKE commands. The GRANT command gives users privileges to base tables and views. The syntax of this command is as follows:

GRANT privileges ON object TO users [WITH GRANT OPTION]

For our purposes object is either a base table or a view. Several privileges can be specified, including these:

SELECT: The right to access (read) all columns of the table specified as the object, including columns added later through ALTER TABLE commands.

INSERT(*column-name*): The right to insert rows with (non-*null* or nondefault) values in the named column of the table named as object. If this right is to be granted with

respect to all columns, including columns that might be added later, we can simply use INSERT. The privileges UPDATE(*column-name*) and UPDATE are similar.

DELETE: The right to delete rows from the table named as object.

REFERENCES(*column-name*): The right to define foreign keys (in other tables) that refer to the speci_ed column of the table object. REFERENCES without a column name speci_ed denotes this right with respect to all columns, including any that are added later.

If a user has a privilege with the grant option, he or she can pass it to another user (with or without the grant option) by using the GRANT command. A user who creates a base table automatically has all applicable privileges on it, along with the right to grant these privileges to other users. A user who creates a view has precisely those privileges on the view that he or she has on *every* one of the view or base tables used to define the view. The user creating the view must have the SELECT privilege on each underlying table, of course, and so is always granted the SELECT privilege on the view. The creator of the view has the SELECT privilege with the grant option only if he or she has the SELECT privilege with the grant option on every underlying table.

In addition, if the view is updatable and the user holds INSERT, DELETE, or UPDATE privileges (with or without the grant option) on the (single) underlying table, the user automatically gets the same privileges on the view.

Only the owner of a schema can execute the data definition statements CREATE, ALTER, and DROP on that schema. The right to execute these statements cannot be granted or revoked.

In conjunction with the GRANT and REVOKE commands, views are an important component of the security mechanisms provided by a relational DBMS. We will discuss the views later in detail. Suppose that user Javed has created the tables COURSE, PROGRAM and STUDENT. Some examples of the GRANT command that Javed can now execute are listed below:

- GRANT INSERT, DELETE ON COURSE TO Puppoo WITH GRANT OPTION
- GRANT SELECT ON COURSE TO Mina
- GRANT SELECT ON PROGRAM TO Mina WITH GRANT OPTION

There is a complementary command to GRANT that allows the withdrawal of privileges. The syntax of the REVOKE command is as follows:

```
REVOKE [GRANT OPTION FOR] privileges ON object FROM users
{RESTRICT | CASCADE}
```

The command can be used to revoke either a privilege or just the grant option on a privilege (by using the optional GRANT OPTION FOR clause). One of the two alternatives, RESTRICT or CASCADE, must be specified; we will see what this choice means shortly. The intuition behind the GRANT command is clear: The creator of a base table or a view is given all the appropriate privileges with respect to it and is allowed to pass these privileges including the right to pass along a privilege to other users. The REVOKE command is, as expected, intended to achieve the reverse: A user who has granted a privilege to another user may change his mind and want to withdraw the granted privilege. The intuition behind exactly what effect a REVOKE command has is complicated by the fact that a user may be granted the same privilege multiple times, possibly by different users.

When a user executes a REVOKE command with the CASCADE keyword, the effect is to withdraw the named privileges or grant option from all users who currently hold these privileges solely through a GRANT command that was previously executed by the same user who is now executing the REVOKE command. If these users received the privileges with the grant option and passed it along, those recipients will also lose their privileges as a consequence of the REVOKE command unless they also received these privileges independently. Consider what happens after the following sequence of commands, where Javed is the creator of COURSE.

GRANT SELECT ON COURSE TO Alia WITH GRANT OPTION (executed by Javed)

GRANT SELECT ON COURSE TO Bobby WITH GRANT OPTION (executed by Alia)

REVOKE SELECT ON COURSSE FROM Alia CASCADE (executed by Javed)

Alia loses the SELECT privilege on COURSE, of course. Then Bobby, who received this privilege from Alia, and only Alia, also loses this privilege. Bobby's privilege is said to be abandoned when the privilege that it was derived from (Alia's SELECT privilege with grant option, in this example) is revoked. When the CASCADE

keyword is specified, all abandoned privileges are also revoked (possibly causing privileges held by other users to become abandoned and thereby revoked recursively). If the RESTRICT keyword is specified in the REVOKE command, the command is rejected if revoking the privileges *just* from the users specified in the command would result in other privileges becoming abandoned.

Consider the following sequence, as another example:

GRANT SELECT ON COURSE TO Alia WITH GRANT OPTION *(executed by Javed)*
GRANT SELECT ON COURSE TO Bobby WITH GRANT OPTION *(executed by Javed)*
GRANT SELECT ON COURSE TO Bobby WITH GRANT OPTION *(executed by Alia)*
REVOKE SELECT ON COURSE FROM Alia CASCADE *(executed by Javed)*

As before, Alia loses the SELECT privilege on COURSE. But what about Bobby? Bobby received this privilege from Alia, but he also received it independently (coincidentally, directly from Javed). Thus Bobby retains this privilege. Consider a third example:

GRANT SELECT ON COURSE TO Alia WITH GRANT OPTION *(executed by Javed)*
GRANT SELECT ON COURSE TO Alia WITH GRANT OPTION *(executed by Javed)*
REVOKE SELECT ON COURSE FROM Alia CASCADE *(executed by Javed)*

Since Javed granted the privilege to Alia twice and only revoked it once, does Alia get to keep the privilege? As per the SQL-92 standard, no. Even if Javed absentmindedly granted the same privilege to Alia several times, he can revoke it with a single REVOKE command. It is possible to revoke just the grant option on a privilege:

GRANT SELECT ON COURSE TO Alia WITH GRANT OPTION *(executed by Javed)*
REVOKE GRANT OPTION FOR SELECT ON COURSE FROM Alia CASCADE *(executed by Javed)*

This command would leave Alia with the SELECT privilege on COURSE, but Alia no longer has the grant option on this privilege and therefore cannot pass it on to other users.

Summary

In this lecture we have studied the different types of joins, with the help of which we can join different tables. We also discussed two major commands of access control that are also considered the part of Data Control Language component of SQL. The last part of this lecture handout is taken from chapter 17 of the reference given. The

interested users are recommended to see the book for detailed discussion on the topic. There is a lot left on SQL, for our course purposes however we have studied enough. Through extensive practice you will have clear understanding that will help you in further learning.

Exercise:

Practice for all various types of Joins and Grant and Revoke commands.

Lecture No. 32

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
Database Management, Jeffery A Hoffer

Overview of Lecture

- Application Programs
- User Interface
- Designing Forms

Until now we have studied conceptual database design, whose goal is to analyze the application and do a conceptual design on the application. The goal is to provide a conceptual design and description of reality. It is independent of data model. Then we discussed the relational database design. A relational database stores all its data inside tables, and nothing more. All operations on data are done on the tables themselves or produce another tables as the result. Here we had discussed the selection of data model. Thereafter we had discussed data manipulation language through SQL. We are using SQL Server as a tool. In this lecture we will discuss application program.

Application Programs

Programs written to perform different requirement posed by the users/organization are the application programs. Application programs can be developed in parallel or after the construction of database design. Tool selection is also critical, but it depends upon developer in which he feels comfortable.

General Activities in Application Programs
Following are the general activities, which are performed during the development of application programs:

- Data input programmes
- Editing
- Display

- Processing related to activities
- Reports

User Interface

In the minds of users, a system's user interface is the system; everything else is just stuff they're happy to ignore. The design of the user interface is therefore critical to the success or failure of a project. Get it right, and your users will forgive the occasional infelicity in implementation. Get it wrong, and it won't really matter how efficient your code is.

The irony here is that if you do get it right, hardly anyone will notice. Really elegant interfaces are invisible. Even if you get it wrong, no one might notice. The interfaces of many computer systems, particularly database systems, are so awkward that your system will be just one more of the mediocre, mildly abusive computer systems people have come to expect.

Effective interfaces can also require more work to implement, although this isn't necessarily the case. In addition, the payoffs can be huge, and they're not all of the "virtue is its own reward" variety. An effective user interface minimizes the time users require to learn and implement the system. Once the system is implemented, productivity gains are higher if users don't have to struggle to bend it to their will. Chances are good that both of these issues were addressed in the project goals. They certainly impact the infamous bottom line.

Effective interfaces that closely match the users' expectations and work processes also minimize the need for external documentation, which is always expensive. And while users might not consciously notice how wonderful your user interface is, they'll certainly notice that your system seems to just work better.

So, what constitutes an effective interface? To my mind, it's one that helps users accomplish their tasks and otherwise gets out of the way. An effective interface doesn't impose its requirements on users. It never forces users to play by its rules; it plays by the users' rules. An effective interface doesn't force users to learn a bunch of

uninteresting stuff just to use it. And finally, it doesn't behave in unexpected ways. Following are the two types of user interfaces:

- Text based
- Graphical User Interface (GUI) most commonly called as Forms

Text Based User Interface

In text based user interface certain keyboard numbers are designated for any action but it is used very rarely nowadays. For example

Adding a Record	1
Deleting a Record	2
Enrollment	3
Result Calculation	4
Exit	5

Forms

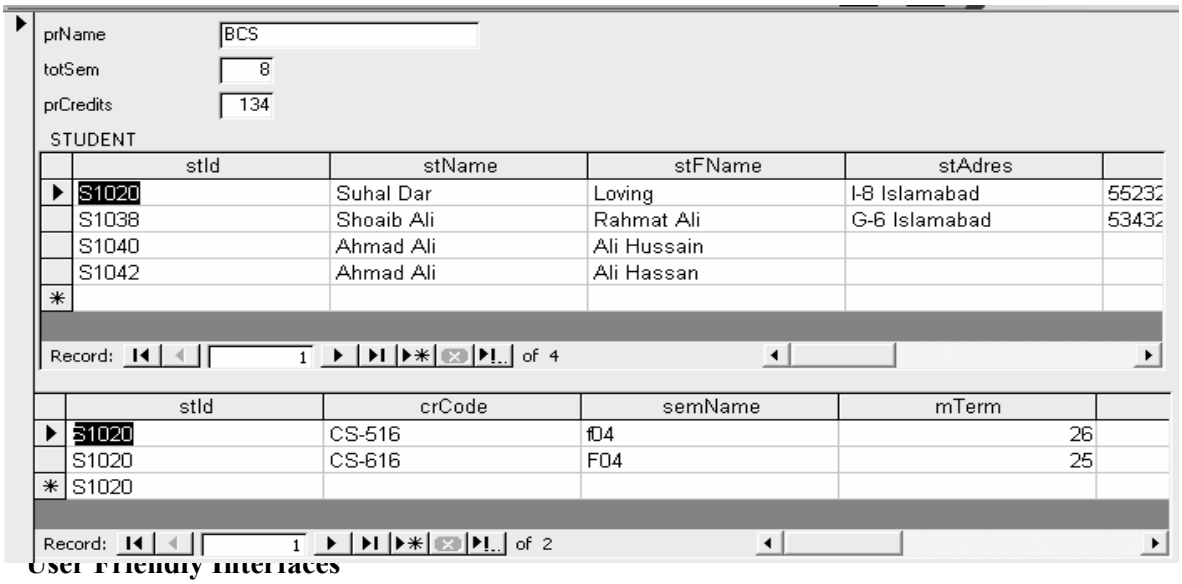
Forms are now days used extensively in the application programs. Following are the different types of forms

Browser Based

These are web-based forms. They are developed in HTML, scripting language or Front Page.

Non-Browser/Simple

Graphical User Interface



User Friendly Interfaces

Two definitions of user-friendly that are often mooted are "easy to learn" and "easy to use." If we put aside for the moment the question of what, exactly, "easy" means, we still have to ask ourselves, "Easy for whom?" The system that's easy for a beginner to learn is not necessarily easy for an expert to use. Your best approach is to consider the needs of each level of user and accommodate each with different facets of the interface. Following are the different kinds of users:

Beginners

Everyone is a beginner at some point. Very few people remain that way—they will either pass through the stage to intermediacy, or they'll discard your system entirely in favor of someone else's. For this reason, you must be careful not to build in support for beginners that will get in the way of more advanced users. Beginners need to know what your system does before they start to learn how to use it. The best way to present this information is outside the main system itself. For simple systems, an introductory dialog box that describes the system can be sufficient. (Just be sure you always include a means of dismissing the dialog box permanently.) For more complex

systems, a guided tour might be more appropriate. Online help isn't a good option for beginners. They might not know it exists or, if they do, how to use it. I have had some success using an online user's guide, however, by including a link to it from the introductory dialog box and from the Help menu. To be successful with beginners, these guides must be task-oriented. Beginners don't want to know what "menu item" means; they want to know how to create an invoice.

Intermediate

For most systems, the majority of users fall into the intermediate category. Intermediate users know what the system does, but they often forget the details of how. This is the group you must support directly in the user interface. Fortunately, the Microsoft Windows interface provides a lot of tools for helping these users.A well-designed menu system is one of the best tools for reminding intermediate users of the system capabilities. A quick scan of the available menu items will immediately remind them of the functions available and at the same time allow them to initiate the appropriate task.

An excellent second level of support for intermediate users is online help. Writing online help is outside the scope of this book. In this context, however, I will mention that most intermediate users will use the index as their primary access mechanism. The index should therefore be as complete as you can possibly make it.

Experts

Expert users know what to do and how to do it. They're primarily interested in doing things quickly. The more shortcuts you can build into your system, the happier you will make this group of users. In my experience, expert users tend to be keyboard-oriented, so make sure that you provide a way to move around the system using the keyboard if you're catering to this group.Expert users also appreciate the ability to customize their working environment. Providing this functionality can be an expensive exercise, however, so you will want to carefully evaluate the benefit before including it. If you do decide to include some level of interface customization, even if it's only a matter of arranging windows on the screen, be certain to maintain the changes between sessions. Nothing is as irritating as having to rearrange everything every time you load a program

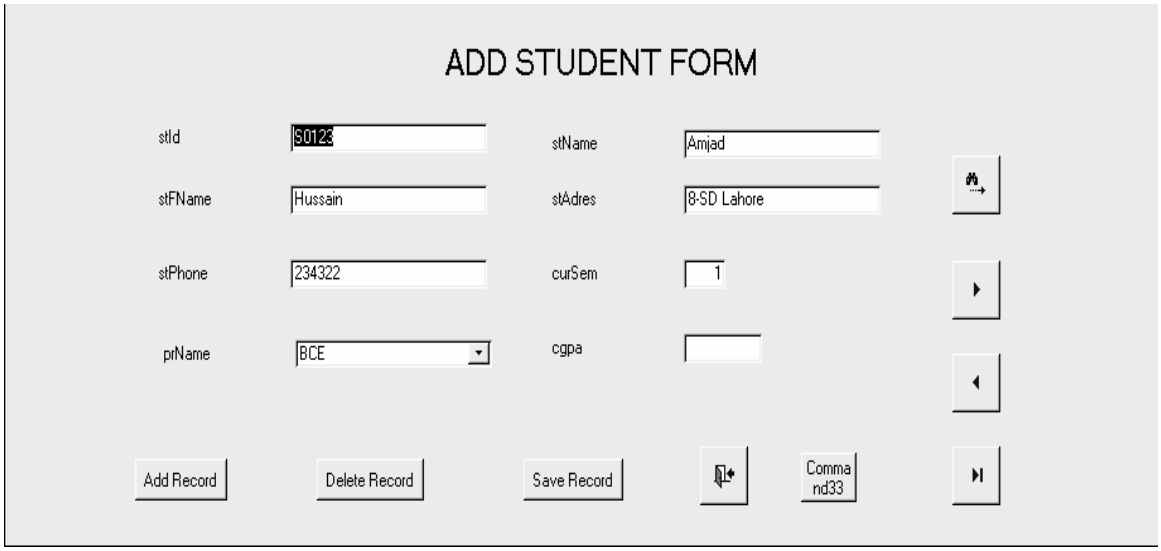
Tips for User Friendly Interface

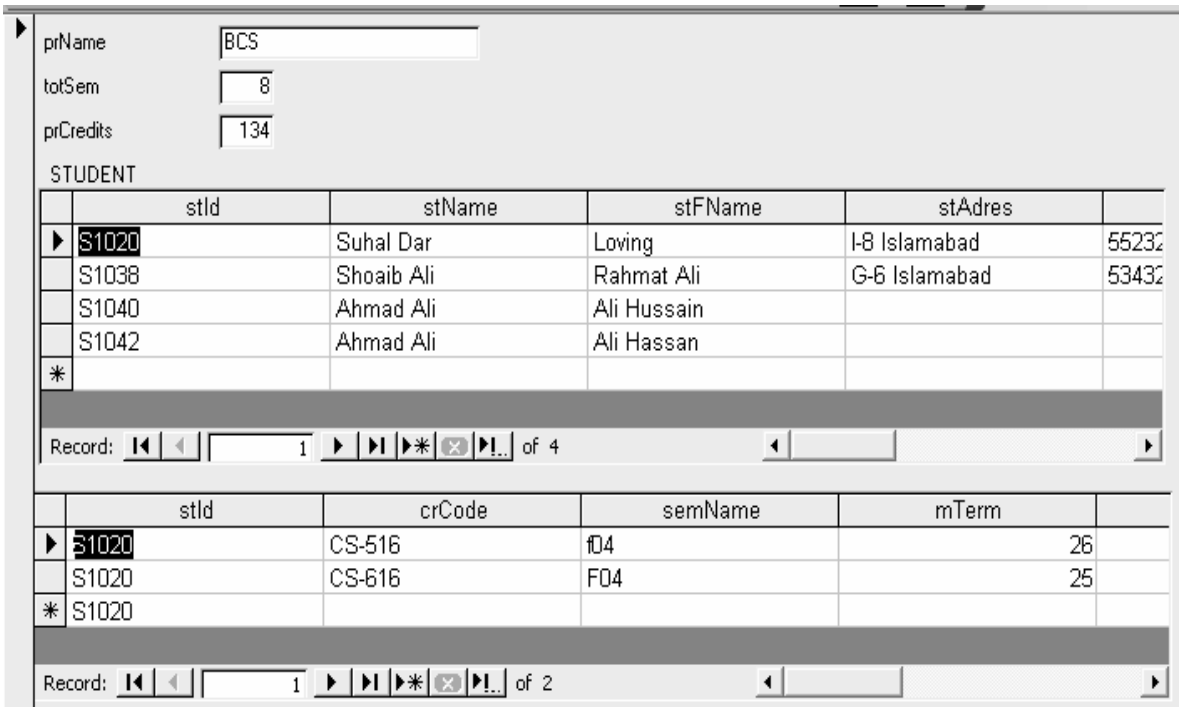
Following are some of the important tips, which should be adhered for interfaces:

- It should be user friendly and user must not search for required buttons or text boxes.
- Interface should be designed in such a manner that user is in charge of the form.
- It should be designed in manner that user memory is always tested
- You should be consistent in your approach while designing interface.
- It should be processes based rather than the data structure based.

Entities and Relationships

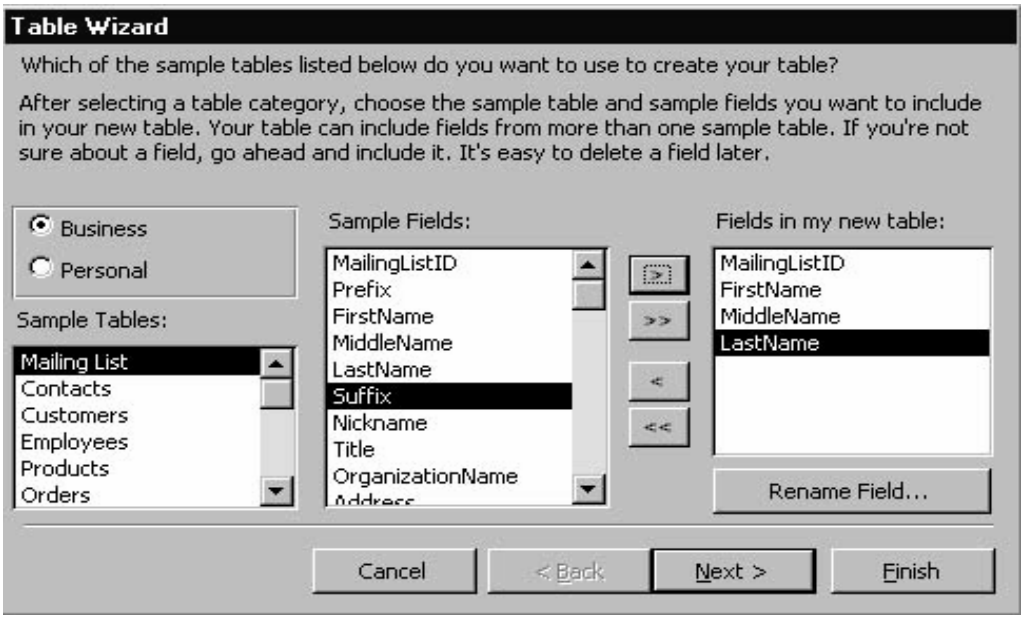
First type is the simple entity on a page.





Windows Controls

There are number of controls which are used to take input and display output like buttons, checkboxes etc. Following are the examples



Numbers, Dates and Text
Normally text boxes are used for the display of dates

Summary

In today's lecture we have studied the application programs and designing user interface and forms. We have studied different techniques and practices for the user interface and form designing. We will discuss an example of form designing in our next lecture.

Lecture No. 33

Reading Material

Programming Microsoft Access
Mastering MS Access

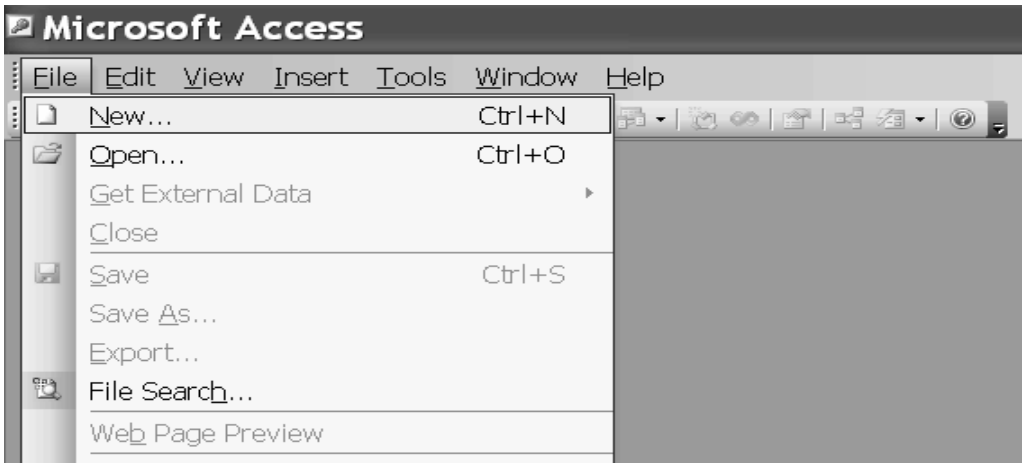
Overview of Lecture

- Designing Input Form
- Arranging Form
- Adding Command Buttons

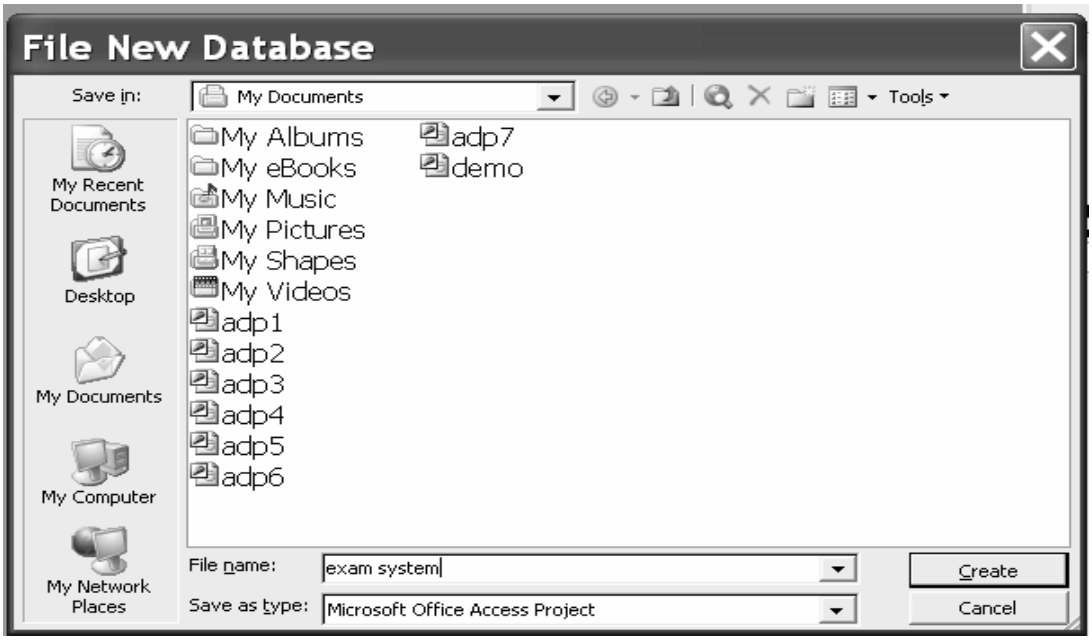
In the previous lecture we have discussed the importance of user interface. It plays an important role in the development of any application. User will take interest in the application if user interface is friendly. We then discussed different tools, which are used in the development of any application. In this lecture we will see the designing of input forms.

An input form is an easy, effective, efficient way to enter data into a table. Input forms are especially useful when the person entering the data is not familiar with the inner workings of Microsoft Access and needs to have a guide in order to input data accurately into the appropriate fields. Microsoft Access provides several predefined forms and provides a forms wizard that walks you through the process of creating a form. One of these predefined forms will be used in the example below. You can also create your own customized forms by using Microsoft Access form design tools. Following things must be ensured for input forms:

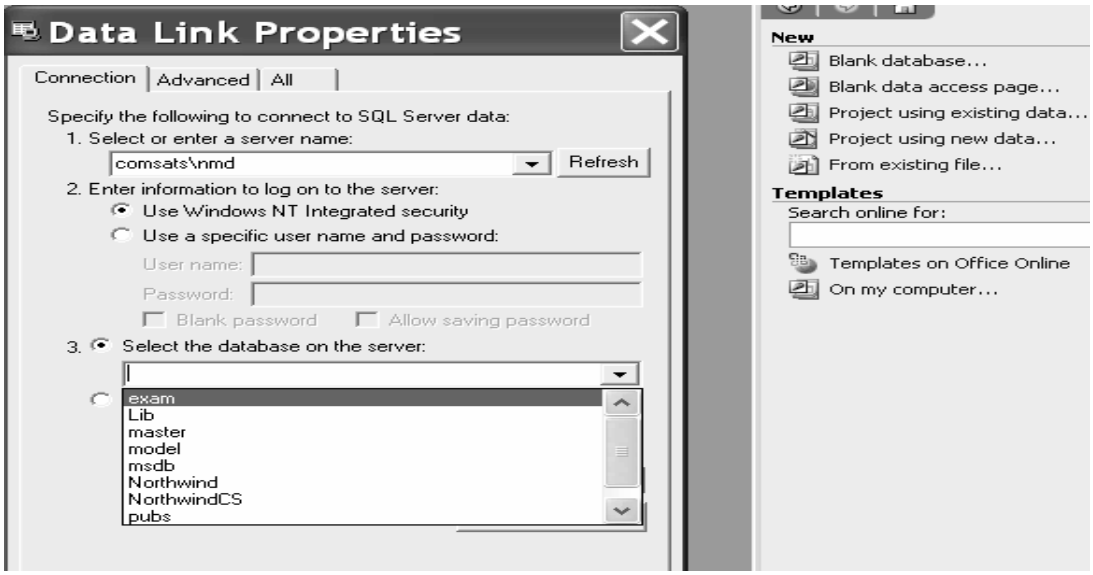
- Forms should be user friendly
- Data integrity must be ensured, which means that database must represent the true picture of real system.
- Checks can be applied within the tables definition or through input forms



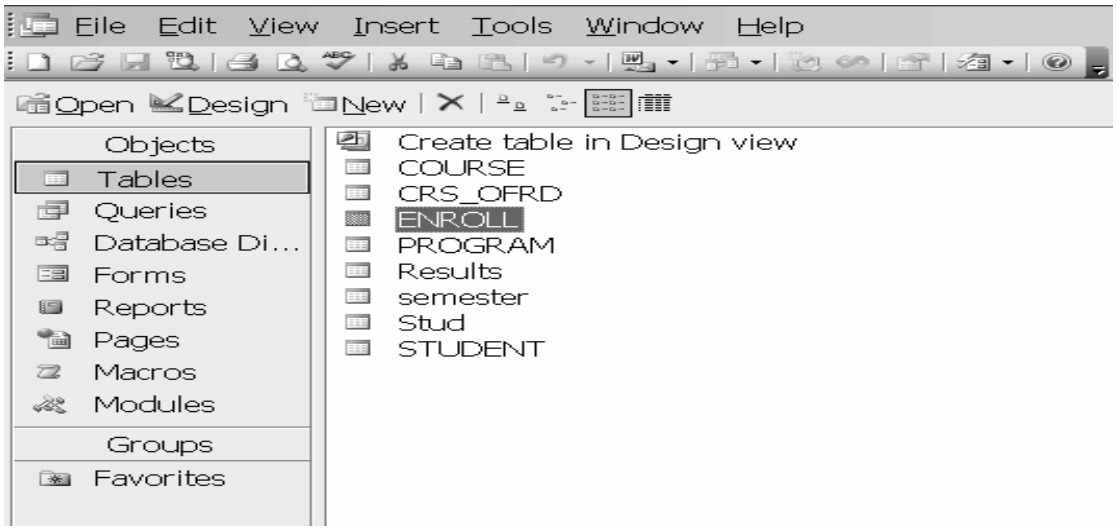
So first we will run MS Access and select New option from the file. Next it will ask the name of database as follows.



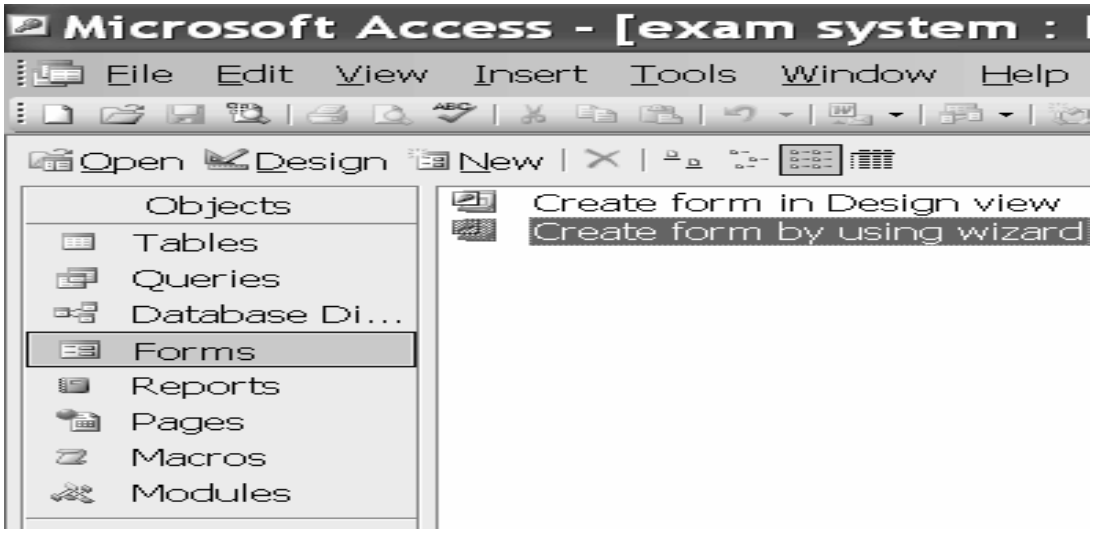
We have given the name as Exam System, we will then press the create button and following screen would be available.



In this screen we will first select project using existing data from New templates, as we are using data of SQL Server that is why we are using this option. Next move on to Data Link Properties dialog, which is adjacent one. So first select the connection Tab in which first select the server name then is the security setting after selecting that option then is the selection of database on which forms are to be constructed.



Now these are the tables of Exam Data base, with which connectivity has been established. Now we will select the Forms option as under:



So here are two options in design and wizard both we are selecting wizard view after selecting this next screen would be as under:

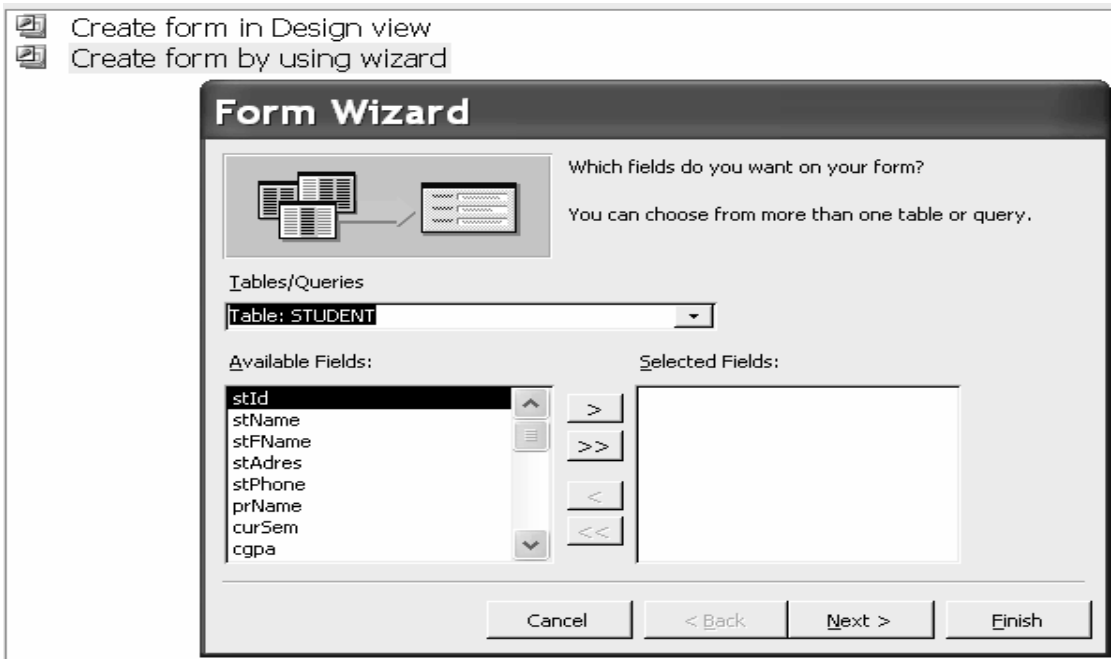
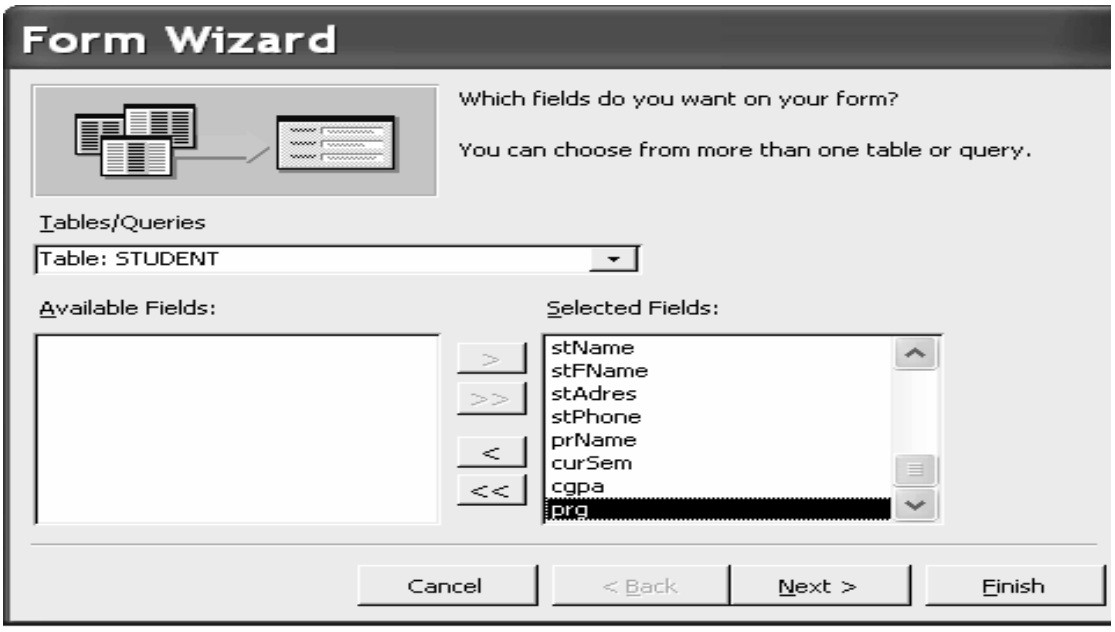
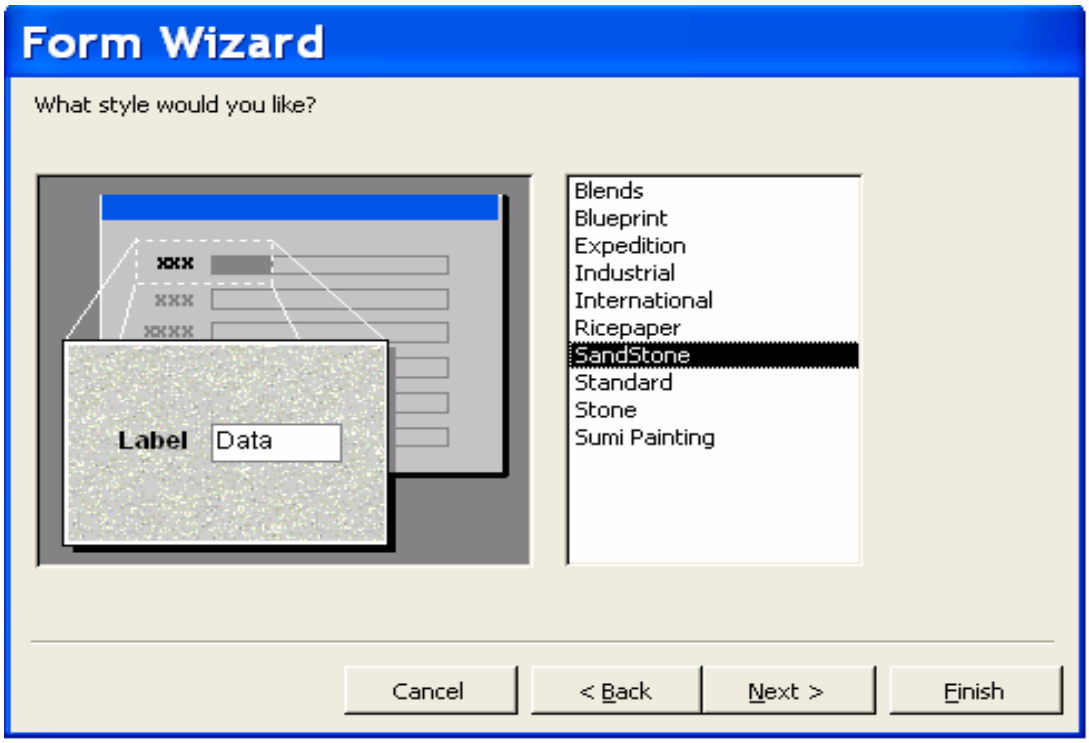


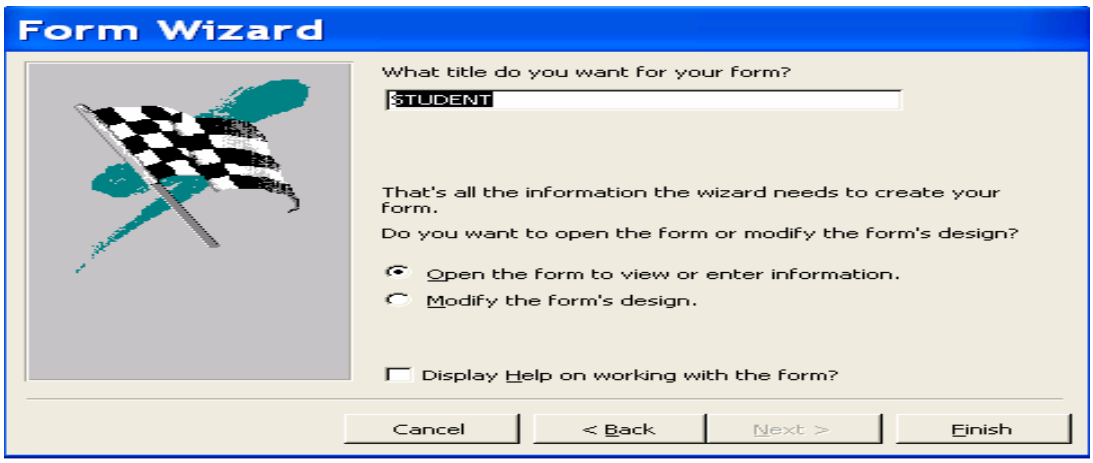
table as under.



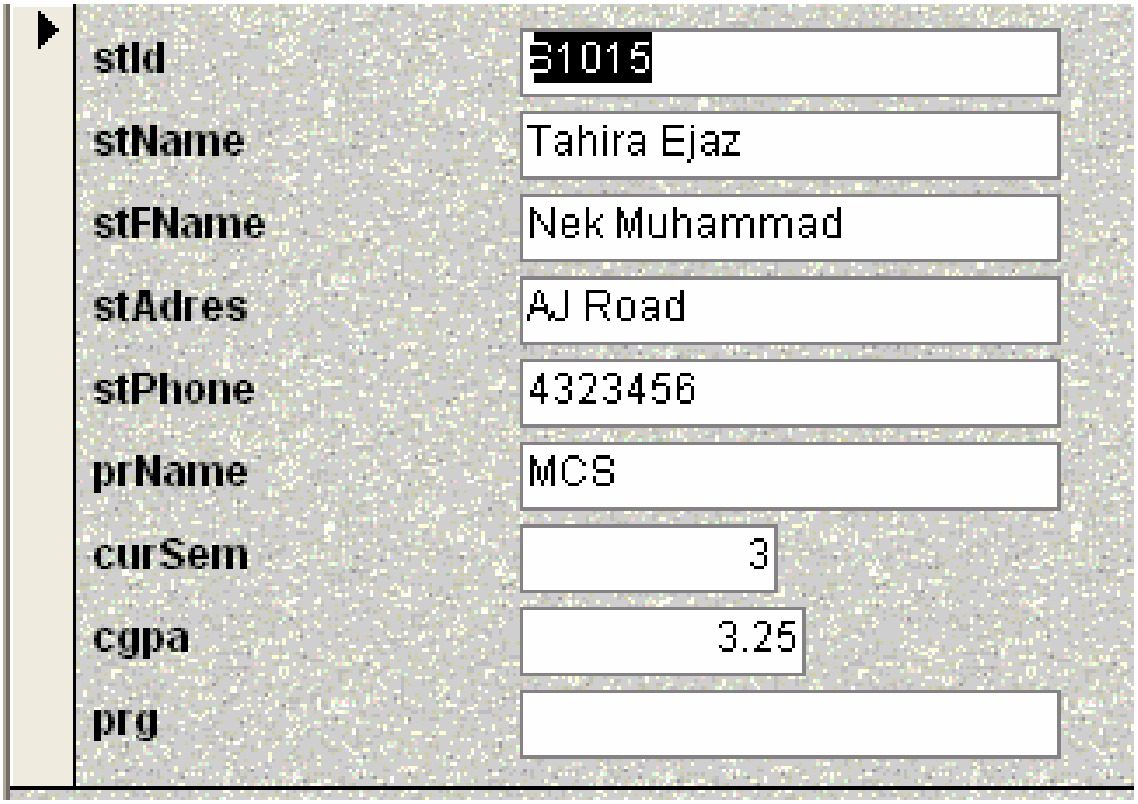
Then is the selection of required layout for the form. There are different options available in this option. We can select the required option. We have selected Column option for the layout of forms. Then is the selection of background or style for our forms as under. We have selected the SandStone option.



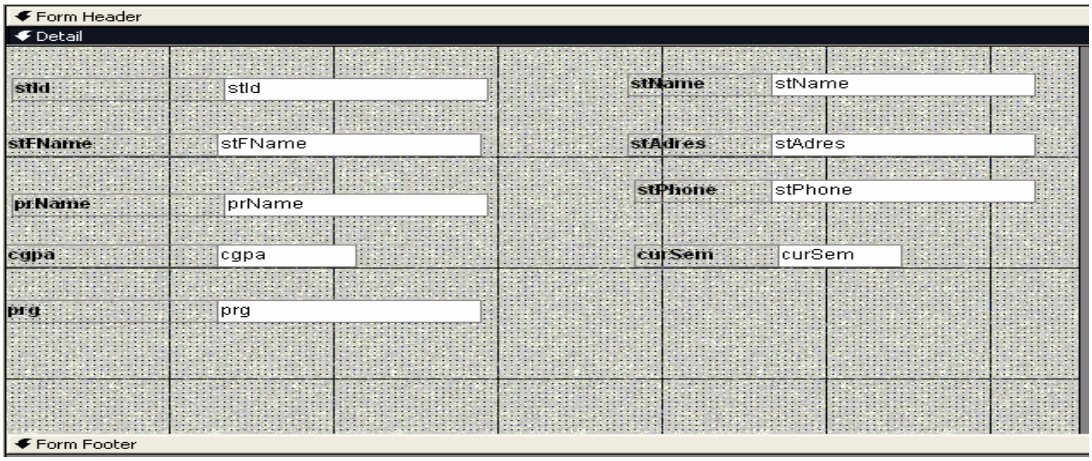
Next is the selection of title for the form as under.



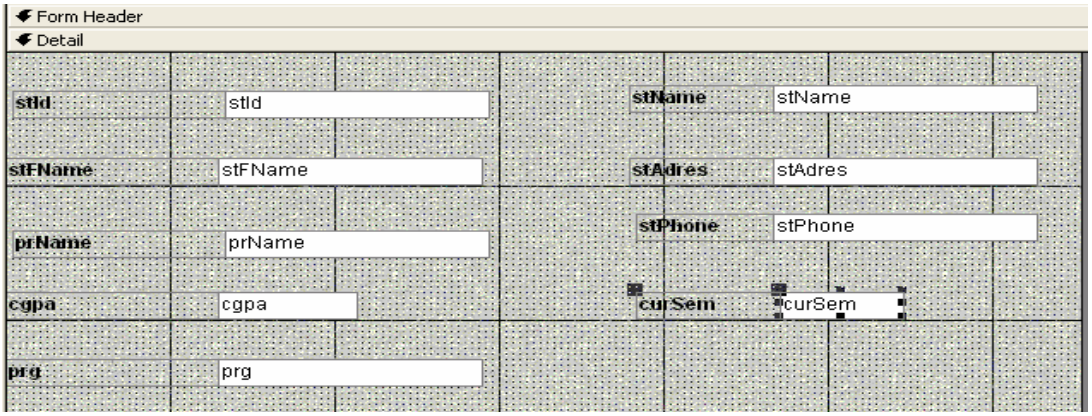
Next is the form view as under in which we can view our data.



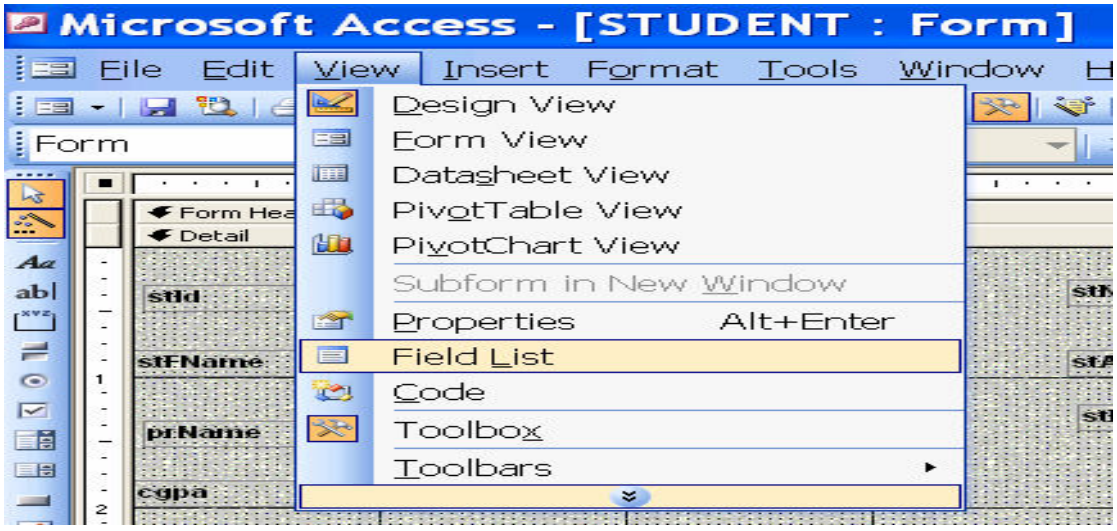
Forms must be designed and arranged in a systematic manner now



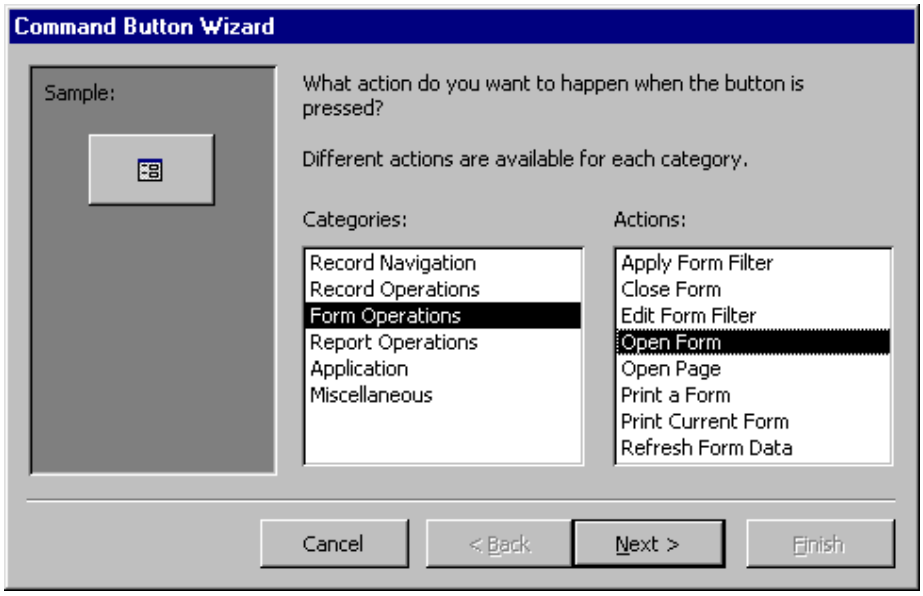
Next is deleting a field from the form.



If we want to add an attribute in the list.



- Open the form in Design View and ensure that the Control Wizard button on the toolbox is pressed in.
- Click the command button icon on the toolbox and draw the button on the form. The Command Button Wizard will then appear.
- On the first dialog window, action categories are displayed in the left list while the right list displays the actions in each category. Select an action for the command button and click Next .



- The next few pages of options will vary based on the action you selected. Continue selecting options for the command button.
- Choose the appearance of the button by entering caption text or selecting a picture. Check the Show All Pictures box to view the full list of available

images. Click **Next >.**

- Enter a name for the command button and click **Finish** to create the button.

In today's lecture we have studied the designing of forms. Forms are used as an alternative way to enter data into a database table. It can be made more perfect with lots of practice and designing number of forms. So it needs practice. We have designed a simple form through wizard. A more complex form can also be designed with some practice.

Lecture No. 34

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
"Modern Database Management", Fred McFadden, Jeffrey Hoffer, Benjamin/Cummings

Overview of Lecture

- Data Storage Concepts
- Physical Storage Media
- Memory Hierarchy

In the previous lecture we have discussed the forms and their designing. From this lecture we will discuss the storage media.

Classification of Physical Storage MediaStorage media are classified according to following characteristics:

Speed of access

Cost per unit of data

Reliability

We can also differentiate storage as either

Volatile storage

Non-volatile storage

Computer storage that is lost when the power is turned off is called as volatile storage. Computer storage that is not lost when the power is turned off is called as non – volatile storage, Pronounced cash a special high-speed storage mechanism. It can be either a reserved section of main memory or an independent high-speed storage device. Two types of caching are commonly used in personal computers: memory caching and disk caching.

A memory cache, sometimes called a cache store or RAM Cache is a portion of memory made of high-speed static RAM (SRAM) instead of the slower and cheaper DRAM used for main memory. Memory caching is effective because most programs access the same data or instructions over and over. By keeping as much of this information as possible in SRAM, the computer avoids accessing the slower DRAM.

Some memory caches are built into the architecture of microprocessors.. The Intel 80486 microprocessor, for example, contains an 8K memory cache, and the Pentium has a 16K cache. Such internal caches are often called Level 1 (L1) caches. Most modern PCs also come with external cache memory, called Level 2 (L2) caches. These caches sit between the CPU and the DRAM. Like L1 caches, L2 caches are composed of SRAM but they are much larger. Disk caching works under the same principle as memory caching, but instead of using high-speed SRAM, a disk cache uses conventional main memory. The most recently accessed data from the disk (as

well as adjacent sectors) is stored in a memory buffer. When a program needs to access data from the disk, it first checks the disk cache to see if the data is there. Disk caching can dramatically improve the performance of applications, because accessing a byte of data in RAM can be thousands of times faster than accessing a byte on a hard disk.

When data is found in the cache, it is called a *cache hit,* and the effectiveness of a cache is judged by its hit rate. Many cache systems use a technique known as smart caching, in which the system can recognize certain types of frequently used data. The strategies for determining which information should be kept in the cache constitute some of the more interesting problems in computer science.

The main memory of the computer is also known as **RAM**, standing for Random Access Memory. It is constructed from integrated circuits and needs to have electrical power in order to maintain its information. When power is lost, the information is lost too! The CPU can directly access it. The access time to read or write any particular byte are independent of whereabouts in the memory that byte is, and currently is approximately 50 **nanoseconds** (a thousand millionth of a second). This is broadly comparable with the speed at which the CPU will need to access data. Main memory is expensive compared to external memory so it has limited capacity. The capacity available for a given price is increasing all the time. For example many home Personal Computers now have a capacity of 16 **megabytes** (million bytes), while 64 megabytes is commonplace on commercial workstations. The CPU will normally transfer data to and from the main memory in groups of two, four or eight bytes, even if the operation it is undertaking only requires a single byte.

Flash memory is a form of EEPROM that allows multiple memory locations to be erased or written in one programming operation. Normal EEPROM only allows one location at a time to be erased or written, meaning that flash can operate at higher effective speeds when the system uses it to read and write to different locations at the same time. All types of flash memory and EEPROM wear out after a certain number of erase operations, due to wear on the insulating oxide layer around the charge storage mechanism used to store data.

Flash memory is non-volatile, which means that it stores information on a silicon chip in a way that does not need power to maintain the information in the chip. This means that if you turn off the power to the chip, the information is retained without consuming any power. In addition, flash offers fast read access times and solid-state shock resistance. These characteristics are why flash is popular for applications such as storage on battery-powered devices like cellular phones and PDAs.Flash memory is based on the Floating-Gate Avalanche-Injection Metal Oxide Semiconductor (FAMOS transistor) which is essentially an NMOS transistor with an additional conductor suspended between the gate and source/drain terminals.

Magnetic disk is round plate on which data can be encoded. There are two basic types of disks: magnetic disks and optical disks on magnetic disks, data is encoded as microscopic magnetized *needles* on the disk's surface. You can record and erase data on a magnetic disk any number of times, just as you can with a cassette tape. Magnetic disks come in a number of different forms:

Floppy Disk: A typical 5¼-inch floppy disk can hold 360K or 1.2MB (megabytes). 3½-inch floppies normally store 720K, 1.2MB or 1.44MB of data.

Hard Disk: Hard disks can store anywhere from 20MB to more than 10GB. Hard disks are also from 10 to 100 times faster than floppy disks.

Optical disks record data by burning microscopic holes in the surface of the disk with a laser. To read the disk, another laser beam shines on the disk and detects the holes by changes in the reflection pattern.
Optical disks come in three basic forms:
CD-ROM: Most optical disks are read-only. When you purchase them, they are already filled with data. You can read the data from a CD-ROM, but you cannot modify, delete, or write new data.
WORM: Stands for **w**rite-**o**nce, **r**ead-**m**any. WORM disks can be written on once and then read any number of times; however, you need a special WORM disk drive to write data onto a WORM disk.
Erasable optical (EO**):** EO disks can be read to, written to, and erased just like magnetic disks.
The machine that spins a disk is called a disk drive. Within each disk drive is one or more *heads* (often called read/write heads) that actually read and write data. Accessing data from a disk is not as fast as accessing data from main memory, but disks are much cheaper. And unlike RAM, disks hold on to data even when the computer is turned off. Consequently, disks are the storage medium of choice for most types of data. Another storage medium is magnetic tape. But tapes are used only for backup and archiving because they are sequential-access devices (to access data in the middle of a tape, the tape drive must pass through all the preceding data).
Short for Redundant Array of Independent (or **I**nexpensive) **D**isks, a category of disk drives that employ two or more drives in combination for fault tolerance and performance. RAID disk drives are used frequently on servers but aren't generally necessary for personal computers.
Fundamental to RAID is "striping", a method of concatenating multiple drives into one logical storage unit. Striping involves partitioning each drive's storage space into stripes which may be as small as one sector (512 bytes) or as large as several megabytes. These stripes are then interleaved round-robin, so that the combined space is composed alternately of stripes from each drive. In effect, the storage space of the drives is shuffled like a deck of cards. The type of application environment, I/O or data intensive, determines whether large or small stripes should be used.
Most multi-user operating systems today, like NT, UNIX and Netware, support overlapped disk I/O operations across multiple drives. However, in order to maximize throughput for the disk subsystem, the I/O load must be balanced across all the drives so that each drive can be kept busy as much as possible. In a multiple drive system without striping, the disk I/O load is never perfectly balanced. Some drives will contain data files which are frequently accessed and some drives will only rarely be accessed. In I/O intensive environments, performance is optimized by striping the drives in the array with stripes large enough so that each record potentially falls entirely within one stripe. This ensures that the data and I/O will be evenly distributed across the array, allowing each drive to work on a different I/O operation, and thus maximize the number of simultaneous I/O operations, which can be performed by the array.
In data intensive environments and single-user systems which access large records, small stripes (typically one 512-byte sector in length) can be used so that each record will span across all the drives in the array, each drive storing part of the data from the record. This causes long record accesses to be performed faster, since the data transfer occurs in parallel on multiple drives. Unfortunately, small stripes rule out multiple overlapped I/O operations, since each I/O will typically involve all drives. However, operating systems like DOS which do not allow overlapped disk I/O, will not be

negatively impacted. Applications such as on-demand video/audio, medical imaging and data acquisition, which utilize long record accesses, will achieve optimum performance with small stripe arrays.

RAID-0
RAID Level 0 is not redundant, hence does not truly fit the "RAID" acronym. In level 0, data is split across drives, resulting in higher data throughput. Since no redundant information is stored, performance is very good, but the failure of any disk in the array results in data loss. This level is commonly referred to as striping.
RAID-1
RAID Level 1 provides redundancy by writing all data to two or more drives. The performance of a level 1 array tends to be faster on reads and slower on writes compared to a single drive, but if either drive fails, no data is lost. This is a good entry-level redundant system, since only two drives are required; however, since one drive is used to store a duplicate of the data, the cost per megabyte is high. This level is commonly referred to as mirroring.
RAID-2
RAID Level 2, which uses Hamming error correction codes, is intended for use with drives which do not have built-in error detection. All SCSI drives support built-in error detection, so this level is of little use when using SCSI drives.

RAID-3
RAID Level 3 stripes data at a byte level across several drives, with parity stored on one drive. It is otherwise similar to level 4. Byte-level striping requires hardware support for efficient use.

RAID-4
RAID Level 4 stripes data at a block level across several drives, with parity stored on one drive. The parity information allows recovery from the failure of any single drive. The performance of a level 4 array is very good for reads (the same as level 0). Writes, however, require that parity data be updated each time. This slows small random writes, in particular, though large writes or sequential writes are fairly fast. Because only one drive in the array stores redundant data, the cost per megabyte of a level 4 array can be fairly low.

RAID-5
RAID Level 5 is similar to level 4, but distributes parity among the drives. This can speed small writes in multiprocessing systems, since the parity disk does not become a bottleneck. Because parity data must be skipped on each drive during reads, however, the performance for reads tends to be considerably lower than a level 4 array. The cost per megabyte is the same as for level 4.
The manner data records are stored and retrieved on physical devices .The technique used to find and retrieve store records are called access methods.

Sequential File Organization
Records are arranged on storage devices in some sequence based on the value of some field, called sequence field. Sequence field is often the key field that identifies the record.

Simply, easy to understand and manage, best for providing sequential access. It is not feasible for direct or random access; inserting/deleting a record in/from the middle of the sequence involves cumbersome record searches and rewriting of the file.
RAID-0 is the fastest and most efficient array type but offers no fault-tolerance.
RAID-1 is the array of choice for performance-critical, fault-tolerant environments. In addition, RAID-1 is the only choice for fault-tolerance if no more than two drives are desired.
RAID-2 is seldom used today since ECC is embedded in almost all modern disk drives.
RAID-3 can be used in data intensive or single-user environments which access long sequential records to speed up data transfer. However, RAID-3 does not allow multiple I/O operations to be overlapped and requires synchronized-spindle drives in order to avoid performance degradation with short records.
RAID-4 offers no advantages over RAID-5 and does not support multiple simultaneous write operations.
RAID-5 is the best choices in multi-user environments which are not write performance sensitive. However, at least three and more typically five drives are required for RAID-5 arrays.

Lecture No. 35

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture

- Hashing
- Hashing Algorithm
- Collision Handling

In the previous lecture we have studied about different storage media and the RAID levels and we started with file organization. In this lecture we will study in length about different types of file organizations.

File Organizations

This is the most common structure for large files that are typically processed in their entirety, and it's at the heart of the more complex schemes. In this scheme, all the records have the same size and the same field format, with the fields having fixed size as well. The records are sorted in the file according to the content of a field of a scalar type, called ``key''. The key must identify uniquely a records, hence different record have diferent keys. This organization is well suited for batch processing of the entire file, without adding or deleting items: this kind of operation can take advantage of the fixed size of records and file; moreover, this organization is easily stored both on disk and tape. The key ordering, along with the fixed record size, makes this organization amenable to dicotomic search. However, adding and deleting records to this kind of file is a tricky process: the logical sequence of records tipycally matches their physical layout on the media storage, so to ease file navigation, hence adding a record and maintaining the key order requires a reorganization of the whole file. The usual solution is to make use of a "log file" (also called "transaction file"), structured as a

pile, to perform this kind of modification, and periodically perform a batch update on the master file.

Sequential files provide access only in a particular sequence. That does not suit many applications since it involves too much time. Some mechanism for direct access is required

Direct Access File Organization:
For most users, the file system is the most visible aspect of an operating system. Files store data and programs. The operating system implements the abstract concept of a file by managing mass storage devices, such as tapes and disks. Also files are normally organized into directories to ease their use, so we look at a variety of directory structures. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. This control is known as file protection. Following are the two types:

- Indexed Sequential
- Direct File Organization

Indexed sequential file:

An index file can be used to effectively overcome the above-mentioned problem, and to speed up the key search as well. The simplest indexing structure is the single-level one: a file whose records are pair's key-pointer, where the pointer is the position in the data file of the record with the given key. Only a subset of data records, evenly spaced along the data file, are indexed, so to mark intervals of data records. A key search then proceeds as follows: the search key is compared with the index ones to find the highest index key preceding the search one, and a linear search is performed from the record the index key points onward, until the search key is matched or until the record pointed by the next index entry is reached. In spite of the double file access (index + data) needed by this kind of search, the decrease in access time with respect to a sequential file is significant.

A type of file access in which an index is used to obtain the address of the block which contains the required record. An index file can be used to effectively to speed up the key search as well. The simplest indexing structure is the single-level one: a

file whose records are pairs key-pointer, where the pointer is the position in the data file of the record with the given key. Only a subset of data records, evenly spaced along the data file, are indexed, so to mark intervals of data records.

A key search then proceeds as follows: the search key is compared with the index ones to find the highest index key preceding the search one, and a linear search is performed from the record the index key points onward, until the search key is matched or until the record pointed by the next index entry is reached. In spite of the double file access (index + data) needed by this kind of search, the decrease in access time with respect to a sequential file is significant.

Consider, for example, the case of simple linear search on a file with 1,000 records. With the sequential organization, an average of 500 key comparisons are necessary (assuming uniformly distributed search key among the data ones). However, using and evenly spaced index with 100 entries, the number of comparisons is reduced to 50 in the index file plus 50 in the data file: a 5:1 reduction in the number of operations.

This scheme can obviously be hyerarchically extended: an index is a sequential file in itself, amenable to be indexed in turn by a second-level index, and so on, thus exploiting more and more the hyerarchical decomposition of the searches to decrease the access time. Obviously, if the layering of indexes is pushed too far, a point is reached when the advantages of indexing are hampered by the increased storage costs, and by the index access times as well.

Indexed:

Why using a single index for a certain key field of a data record? Indexes can be obviously built for each field that uniquely identifies a record (or set of records within the file), and whose type is amenable to ordering. Multiple indexes hence provide a high degree of flexibility for accessing the data via search on various attributes; this organization also allows the use of variable length records (containing different fields).

It should be noted that when multiple indexes are are used the concept of sequentiality of the records within the file is useless: each attribute (field) used to construct an index typically imposes an ordering of its own. For this very reason is typicaly not possible to use the ``sparse'' (or ``spaced'') type of indexing previously described. Two

types of indexes are usually found in the applications: the exhaustive type, which contains an entry for each record in the main file, in the order given by the indexed key, and the partial type, which contain an entry for all those records that contain the chosen key field (for variable records only).

Defining Keys:

An indexed sequential file must have at least one key. The first (primary) key is always numbered 0. An indexed sequential file can have up to 255 keys; however, for file-processing efficiency it is recommended that you define no more than 7 or 8 keys. (The time required to insert a new record or update an existing record is directly related to the number of keys defined; the retrieval time for an existing record, however, is unaffected by the number of keys.)

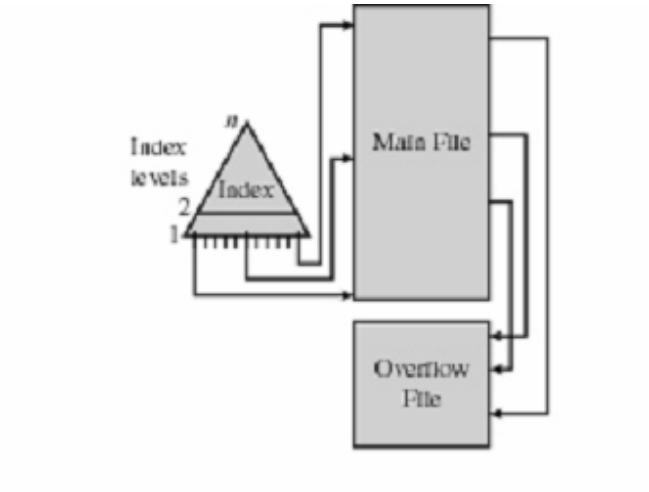
When you design an indexed sequential file, you must define each key in the following terms:

- Position and size
- Data type
- Index number
- Options selected

When you want to define more than one key, or to define keys of different data types, you must be careful when you specify the key fields. The next few subsections describe some considerations for specifying keys. In Indexed sequential files following are ensured:

- New records are added to an overflow file
- Record in main file that precedes it is updated to contain a pointer to the new record
- The overflow is merged with the main file during a batch update
- Multiple indexes for the same key field can be set up to increase efficiency

The diagram of Index sequential file is as under:



Indexed Sequential Summary:
Following are salient features of Indexed sequential file structure:
Records are stored in sequence and index is maintained.
Dense and nondense types of indexes are maintained.
Track overflows and file overflow areas are ensured.
Cylinder index increases the efficiency .

Lecture No. 36

Reading Material

"Database Management Systems", 2nd edition, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill
Database Management, Jeffery A Hoffer

Overview of Lecture

- Hashing
- Hashing Algorithm
- Collision Handling

In the previous lecture we have discussed file organization and techniques of file handling. In today's lecture we will study hashing techniques, its algorithms and collision handling.

Hashing

A hash function is computed on some attribute of each record. The result of the function specifies in which block of the file the record should be placed .Hashing provides rapid, non-sequential, direct access to records. A key record field is used to calculate the record address by subjecting it to some calculation; a process called hashing. For numeric ascending order a sequential key record fields this might involve simply using relative address indexes from a base storage address to access records. Most of the time, key field does not have the values in sequence that can directly be used as relative record number. It has to be transformed. Hashing involves computing the address of a data item by computing a function on the search key value.

A hash function h is a function from the set of all search key values *K* to the set of all bucket addresses *B*.

- We choose a number of buckets to correspond to the number of search key values we will have stored in the database.
- To perform a lookup on a search key value K_i, we compute $h(K_i)$, and search the bucket with that address.
- If two search keys *i* and *j* map to the same address, because $h(K_i) = h(K_j)$, then the bucket at the address obtained will contain records with both search key values.
- In this case we will have to check the search key value of every record in the bucket to get the ones we want.
- Insertion and deletion are simple.

Hash Functions

A good hash function gives an average-case lookup that is a small constant, independent of the number of search keys. We hope records are distributed uniformly among the buckets. The worst hash function maps all keys to the same bucket. The best hash function maps all keys to distinct addresses. Ideally, distribution of keys to addresses is uniform and random.

Hashed Access Characteristics

Following are the major characteristics:

- No indexes to search or maintain
- Very fast direct access
- Inefficient sequential access
- Use when direct access is needed, but sequential access is not.

Mapping functions

The direct address approach requires that the function, h(k), is a one-to-one mapping from each k to integers in (1,m). Such a function is known as a perfect hashing function: it maps each key to a distinct integer within some manageable range and enables us to trivially build an O(1) search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a hash function, h(k), which maps most of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of collisions

(cases where multiple keys map onto the same integer), is sufficiently small, then hash tables work quite well and give O(1) search times.

Handling the Collisions

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element, which should be stored in a single slot of the table. Various techniques are used to manage this problem:

- Chaining,
- Overflow areas,
- Re-hashing,
- Using neighboring slots (linear probing),
- Quadratic probing,
- Random probing, ...

Chaining:

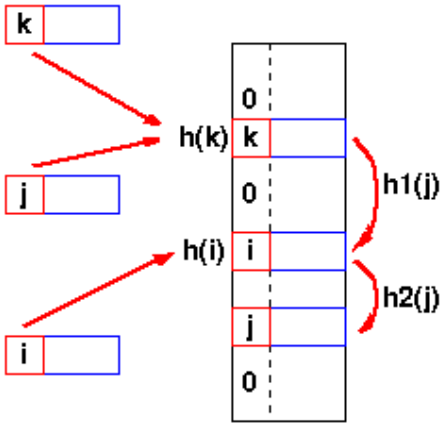
One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Re-hashing:

Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found. The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.

Linear probing:

One of the simplest re-hashing functions is +1 (or -1), ie on a collision; look in the neighboring slot in the table. It calculates the new address extremely quickly and may be extremely efficient on a modern RISC processor due to efficient cache utilization. The animation gives you a practical demonstration of the effect of linear probing: it also implements a quadratic re-hash function so that you can compare the difference.



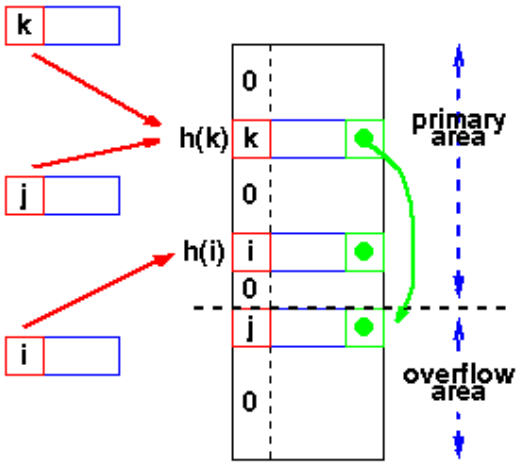
h(j)=h(k), so the next hash function, h1 is used. A second collision occurs, so h2 is used.

Clustering:

Linear probing is subject to a clustering phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.

Overflow area:

Another scheme will divide the pre-allocated table into two sections: the primary area to which keys are mapped and an area for collisions, normally termed the overflow area.



When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.

Collision handling

A well-chosen hash function can avoid anomalies which are result of an excessive number of collisions, but does not eliminate collisions. We need some method for handling collisions when they occur. We'll consider the following techniques:

- Open addressing
- Chaining

Open addressing:

The hash table is an array of (key, value) pairs. The basic idea is that when a (key, value) pair is inserted into the array, and a collision occurs, the entry is simply inserted at an alternative location in the array. Linear probing, double hashing, and rehashing are all different ways of choosing an alternative location. The simplest probing method is called linear probing. In linear probing, the probe sequence is simply the sequence of consecutive locations, beginning with the hash value of the key. If the end of the table is reached, the probe sequence wraps around and continues at location 0. Only if the table is completely full will the search fail.

Summary Hash Table Organization:

Organization	Advantages	Disadvantages
Chaining	Unlimited number of elements Unlimited number of collisions	Overhead of multiple linked lists
Re-hashing	Fast re-hashing Fast access through use of main table space	Maximum number of elements must be known Multiple collisions may become probable
Overflow area	Fast access Collisions don't use primary table space	Two parameters which govern performance need to be estimated

Lecture No. 37

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture:

- Indexes
- Index Classification

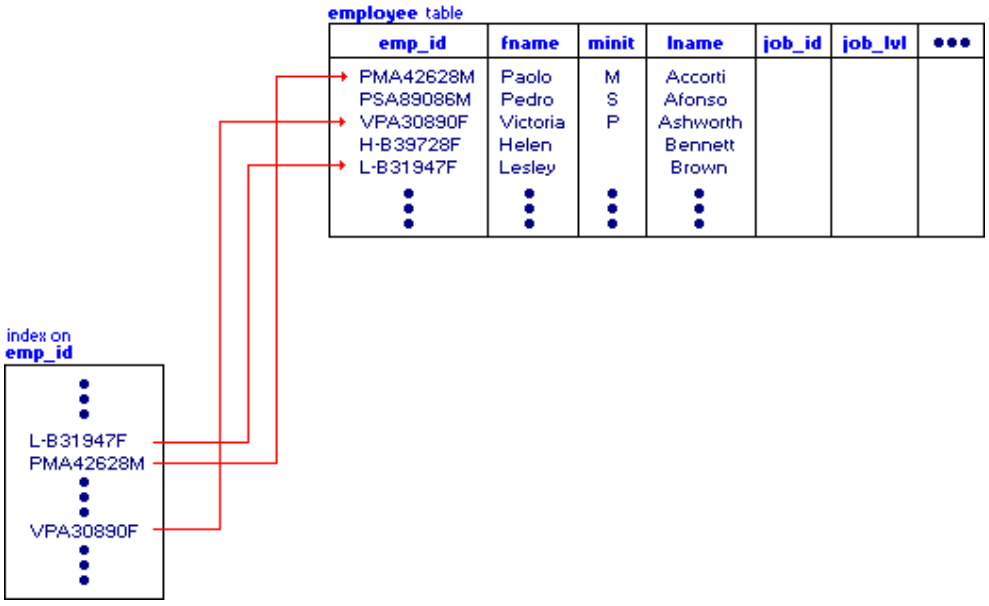
In the previous lecture we have discussed hashing and collision handling. In today's lecture we will discuss indexes, their properties and classification.

Index

In a book, the index is an alphabetical listing of topics, along with the page number where the topic appears. The idea of an INDEX in a Database is similar. We will consider two popular types of indexes, and see how they work, and why they are useful. Any subset of the fields of a relation can be the search key for an index on the relation. Search key is not the same as k*ey* (e.g. doesn't have to be unique ID). An index contains a collection of data entries, and supports efficient retrieval of all records with a given search key value k. typically, index also contains auxiliary information that directs searches to the desired data entries. There can be multiple (different) indexes per file. Indexes on primary key and on attribute(s) in the unique constraint are automatically created. Indexes in databases are similar to indexes in books. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a book is a list of words with the page numbers that contain each word. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value. Indexes can be

created on either a single column or a combination of columns in a table and are implemented in the form of B-trees. An index contains an entry with one or more columns (the search key) from each row in a table. A B-tree is sorted on the search key, and can be searched efficiently on any leading subset of the search key. For example, an index on columns A, B, C can be searched efficiently on A, on A, B, and A, B, C. Most books contain one general index of words, names, places, and so on. Databases contain individual indexes for selected types or columns of data: this is similar to a book that contains one index for names of people and another index for places. When you create a database and tune it for performance, you should create indexes for the columns used in queries to find data.

In the pubs sample database provided with Microsoft® SQL Server™ 2000, the employee table has an index on the emp_id column. The following illustration shows that how the index stores each emp_id value and points to the rows of data in the table with each value.



When SQL Server executes a statement to find data in the employee table based on a specified emp_id value, it recognizes the index for the emp_id column and uses the index to find the data. If the index is not present, it performs a full table scan starting at the beginning of the table and stepping through each row, searching for the specified emp_id value. SQL Server automatically creates indexes for certain types of

constraints (for example, PRIMARY KEY and UNIQUE constraints). You can further customize the table definitions by creating indexes that are independent of constraints.

The performance benefits of indexes, however, do come with a cost. Tables with indexes require more storage space in the database. Also, commands that insert, update, or delete data can take longer and require more processing time to maintain the indexes. When you design and create indexes, you should ensure that the performance benefits outweigh the extra cost in storage space and processing resources.

Index Classification

Indexes are classified as under:

- Clustered vs. Un-clustered Indexes
- Single Key vs. Composite Indexes
- Tree-based, inverted files, pointers

Primary Indexes:

Consider a table, with a Primary Key Attribute being used to store it as an ordered array (that is, the records of the table are stored in order of increasing value of the Primary Key attribute.)As we know, each BLOCK of memory will store a few records of this table. Since all search operations require transfers of complete blocks, to search for a particular record, we must first need to know which block it is stored in. If we know the address of the block where the record is stored, searching for the record is very fast. Notice also that we can order the records using the Primary Key attribute values. Thus, if we just know the primary key attribute value of the first record in each block, we can determine quite quickly whether a given record can be found in some block or not. This is the idea used to generate the Primary Index file for the table.

Secondary Indexes:

Users often need to access data on the basis of non-key or non-unique attribute; secondary key. Like student name, program name, students enrolled in a particular program .Records are stored on the basis of key attribute; three possibilities

- Sequential search
- Sorted on the basis of name

- Sort for command execution

A part from primary indexes, one can also create an index based on some other attribute of the table. We describe the concept of Secondary Indexes for a Key attribute that is, for an attribute which is not the Primary Key, but still has unique values for each record of the table The idea is similar to the primary index. However, we have already ordered the records of our table using the Primary key. We cannot order the records again using the secondary key (since that will destroy the utility of the Primary Index Therefore, the Secondary Index is a two column file, which stores the address of every tuple of the table.

Following are the three-implementation approaches of Indexes:

- Inverted files or inversions
- Linked lists
- B+ Trees

Creating Index

CREATE [UNIQUE]

[CLUSTERED | NONCLUSTERED]

INDEX *index_name*

ON { *table* | *view* } **(** *column* [ASC | DESC] [**,**...*n*]

We will now see an example of creating index as under:

CREATE UNIQUE INDEX pr_prName

ON program(prName)

It can also be created on composite attributes. We will see it with an example.

CREATE UNIQUE INDEX

St_Name ON

Student(stName ASC, stFName DESC)

Properties of Indexes:

Following are the major properties of indexes:

- Indexes can be defined even when there is no data in the table
- Existing values are checked on execution of this command
- It support selections of form as under:

  field <operator> constant

- It support equality selections as under:
  Either “tree” or “hash” indexes help here.
- It support Range selections (operator is one among <, >, <=, >=, BETWEEN)

Summary

In today’s lecture we have discussed the indexes, its classification and creating the indexes as well. The absence or presence of an index does not require a change in the wording of any SQL statement. An index merely offers a fast access path to the data; it affects only the speed of execution. Given a data value that has been indexed, the index points directly to the location of the rows containing that value. Indexes are logically and physically independent of the data in the associated table. You can create or drop an index at anytime without affecting the base tables or other indexes. If you drop an index, all applications continue to work; however, access to previously indexed data might be slower. Indexes, being independent structures, require storage space.

Lecture No. 38

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture

Indexes
Clustered Versus Un-clustered Indexes
Dense Verses Sparse Indexes
Primary and Secondary Indexes
Indexes Using Composite Search Keys

In the previous lecture we studied about what the indexes are and what is the need for creating an index. There exist a number of index types which are important and are helpful for the file organization in database environments for the storage of files on disks.

Ordered Indices

In order to allow fast **random** access, an index structure may be used.
A file may have several indices on different search keys.
If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering index**.
Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

Clustered Indexes

A **clustered index** determines the storage order of data in a table. A clustered index is analogous to a telephone directory, which arranges data by last name. Because the clustered index dictates the physical storage order of the data in the table, a table can contain only one clustered index. However, the index can comprise multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.
A clustered index is particularly efficient on columns often searched for ranges of values. Once the row with the first value is found using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if an application frequently executes a query to retrieve records between a range of dates, a clustered index can quickly locate the row containing the beginning date, and then retrieve all adjacent rows in the table until the last date is reached. This can help increase the performance of this type of query. Also, if there is a column(s) which is used frequently to sort the data retrieved from a table, it can be advantageous to

cluster (physically sort) the table on that column(s) to save the cost of a sort each time the column(s) is queried.
Clustered indexes are also efficient for finding a specific row when the indexed value is unique. For example, the fastest way to find a particular employee using the unique employee ID column **emp_id** would be to create a clustered index or PRIMARY KEY constraint on the **emp_id** column.
Note PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.
Alternatively, a clustered index could be created on **lname, fname** (last name, first name), because employee records are often grouped and queried that way rather than by employee ID.

Non-clustered Indexes

Nonclustered indexes have the same B-tree structure as clustered indexes, with two significant differences:
The data rows are not sorted and stored in order based on their nonclustered keys.
The leaf layer of a nonclustered index does not consist of the data pages.
Instead, the leaf nodes contain index rows. Each index row contains the nonclustered key value and one or more row locators that point to the data row (or rows if the index is not unique) having the key value.
Nonclustered indexes can be defined on either a table with a clustered index or a heap.
In Microsoft® SQL Server™ version 7.0, the row locators in nonclustered index rows have two forms:
If the table is a heap (does not have a clustered index), the row locator is a pointer to the row. The pointer is built from the file ID, page number, and number of the row on the page. The entire pointer is known as a Row ID.
If the table does have a clustered index, the row locator is the clustered index key for the row. If the clustered index is not a unique index, SQL Server 7.0 adds an internal value to duplicate keys to make them unique. This value is not visible to users; it is used to make the key unique for use in nonclustered indexes. SQL Server retrieves the data row by searching the clustered index using the clustered index key stored in the leaf row of the nonclustered index.
Because nonclustered indexes store clustered index keys as their row locators, it is important to keep clustered index keys as small as possible. Do not choose large columns as the keys to clustered indexes if a table also has nonclustered indexes.

Dense and Sparse Indices

There are Two types of ordered indices:

Dense Index:

An index record appears for every search key value in file.
This record contains search key value and a pointer to the actual record.

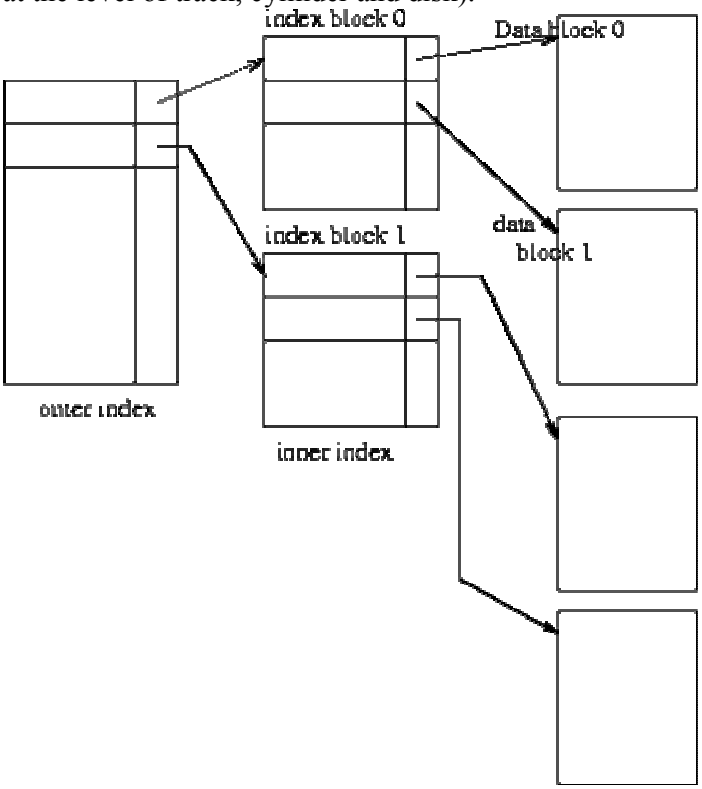
Sparse Index:

Index records are created only for some of the records.
To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.
Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions.
We can have a good compromise by having a sparse index with one entry per block. It has several advantages.

Biggest cost is in bringing a block into main memory.
We are guaranteed to have the correct block with this method, unless record is on an overflow block (actually could be several blocks).
Index size still small.

Multi-Level Indices

Even with a sparse index, index size may still grow too large. For 100,000 records, 10 per block, at one index record per block, that's 10,000 index records! Even if we can fit 100 index records per block, this is 100 blocks.
If index is too large to be kept in main memory, a search results in several disk reads.
If there are no overflow blocks in the index, we can use binary search.
This will read as many as $1 + \log_2(b)$ blocks (as many as 7 for our 100 blocks).
If index has overflow blocks, then sequential search typically used, reading **all** *b* index blocks.
Use binary search on outer index. Scan index block found until correct index record found. Use index record as before - scan block pointed to for desired record.
For very large files, additional levels of indexing may be required.
Indices must be updated at all levels when insertions or deletions require it.
Frequently, each level of index corresponds to a unit of physical storage (e.g. indices at the level of track, cylinder and disk).



Two-level sparse index.

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

Deletion:

Find (look up) the record
If the last record with a particular search key value, delete that search key value from index.

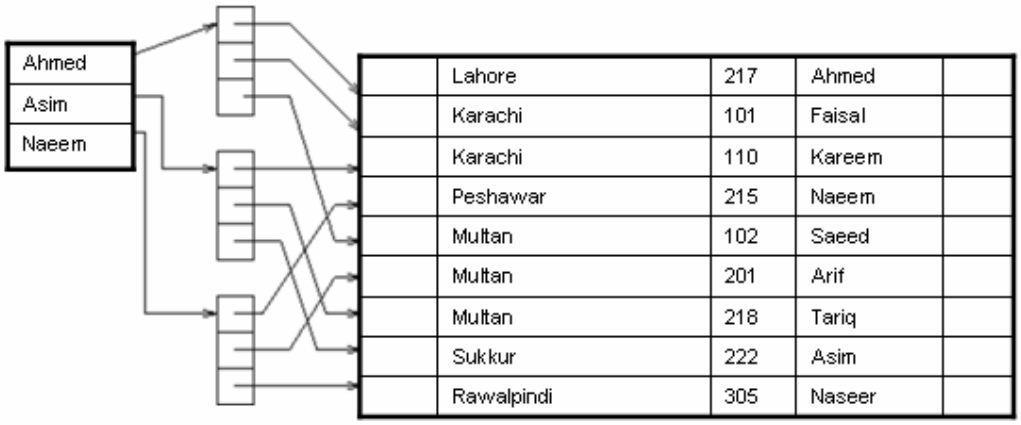
For dense indices, this is like deleting a record in a file.
For sparse indices, delete a key value by replacing key value's entry in index by next search key value. If that value already has an index entry, delete the entry.

Insertion:

Find place to insert.
Dense index: insert search key value if not present.
Sparse index: no change unless new block is created. (In this case, the first search key value appearing in the new block is inserted into the index).

Secondary Indices

If the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value because the remaining records with the same search-key value could be anywhere in the file. Therefore, a secondary index must contain pointers to all the records.

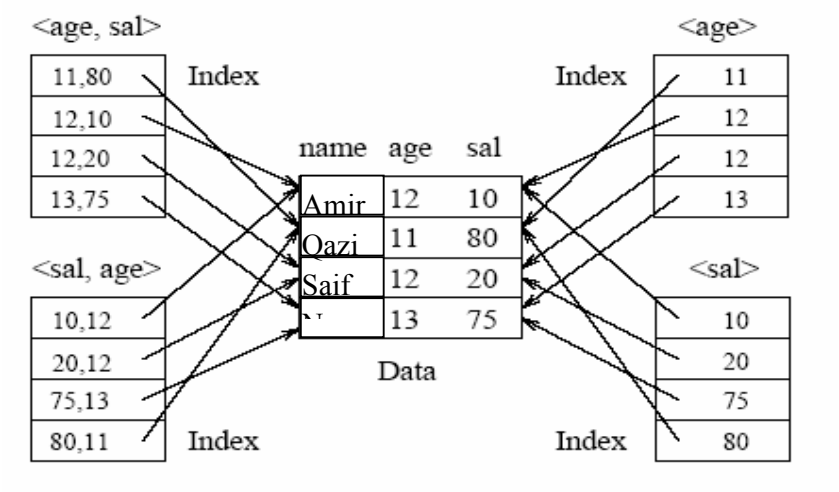


Sparse secondary index on *sname*.
We can use an extra-level of indirection to implement secondary indices on search keys that are not candidate keys. A pointer does not point directly to the file but to a bucket that contains pointers to the file.
See Figure above on secondary key *sname*.
To perform a lookup on Peterson, we must read all three records pointed to by entries in bucket 2.
Only one entry points to a Peterson record, but three records need to be read.
As file is not ordered physically by *cname*, this may take 3 block accesses.
Secondary indices must be **dense**, with an index entry for every search-key value, and a pointer to every record in the file.
Secondary indices improve the performance of queries on non-primary keys.
They also impose serious overhead on database modification: whenever a file is updated, every index must be updated.
Designer must decide whether to use secondary indices or not.

Indexes Using Composite Search Keys

The search key for an index can contain several fields; such keys are called **composite search keys** or **concatenated keys**. As an example, consider a collection of employee records, with fields *name, age*, and *sal*, stored in sorted order by *name*. Figure below illustrates the difference between a composite index with key (*age, sal) and* a

composite index with key (*sal, age)*, an index with key *age*, and an index with key *sal*. All indexes which are shown in the figure use Alternative (2) for data entries.



If the search key is composite, an equality query is one in which each field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with age = 20 and sal = 10. The hashed file organization supports only equality queries, since a hash function identifies the bucket containing desired records only if a value is specified for each field in the search key. A range query is one in which not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with age = 20; this query implies that any value is acceptable for the sal field. As another example of a range query, we can ask to retrieve all data entries with age < 30 and sal > 40.

Lecture No. 39 and 40

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture

- Introduction to Views
- Views, Data Independence, Security
- Choosing a Vertical and Horizontal Subset of a Table
- A View Using Two Tables
- A View of a View
- A View Using a Function
- Updates on Views

Views

Views are generally used to focus, simplify, and customize the perception each user has of the database. Views can be used as security mechanisms by allowing users to access data through the view, without granting the users permissions to directly access the underlying base tables of the view.

To Focus on Specific Data

Views allow users to focus on specific data that interests them and on the specific tasks for which they are responsible. Unnecessary data can be left out of the view. This also increases the security of the data because users

can see only the data that is defined in the view and not the data in the underlying table.

A database view displays one or more database records on the same page. A view can display some or all of the database fields. Views have filters to determine which records they show. Views can be sorted to control the record order and grouped to display records in related sets. Views have other options such as totals and subtotals.

Most users interact with the database using the database views. A key to creating a useful database is a well-chosen set of views. Luckily, while views are powerful, they are also easy to create.

A "view" is essentially a dynamically generated "result" table that is put together based upon the parameters you have defined in your query. For example, you might instruct the database to give you a list of all the employees in the EMPLOYEES table with salaries greater than 50,000 USD per year. The database would check out the EMPLOYEES table and return the requested list as a "virtual table".

Similarly, a view could be composed of the results of a query on several tables all at once (sometimes called a "join"). Thus, you might create a view of all the employees with a salary of greater than 50K from several stores by

accumulating the results from queries to the EMPLOYEES and STORES databases. The possibilities are limitless.

You can customize all aspects of a view, including:

- The name of the view
- The fields that appear in the view
- The column title for each field in the view
- The order of the fields in the view
- The width of columns in the view, as well as the overall width of the view
- The set of records that appear in the view (Filtering)
- The order in which records are displayed in the view (Sorting & Grouping)
- Column totals for numeric and currency fields (Totaling & Subtotaling)

The physical schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The conceptual schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, i.e., be part of the external schema of the database, additional relations in the external schema can be defined using the view mechanism. The view mechanism thus provides the support for logical data independence in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we can define a view with the old schema, and applications that expect to see the old schema can now use this view. Views are also valuable in the context of security: We can define views

that give a group of user's access to just the information they are allowed to see. For example, we can define a view that allows students to see other students' name and age but not their GPA, and allow all students to access this view, but not the underlying Students table.

There are two ways to create a new view in your database. You can:

- Create a new view from scratch.
- Or, make a copy of an existing view and then modify it.

Characteristics /Types of Views:

We have a number of views type of which some of the important views types are listed below:

- **Materialized View**
- **Simple Views**
- **Complex View**
- **Dynamic Views.**

A materialized view is a replica of a target master from a single point in time. The master can be either a master table at a master site or a master materialized view at a materialized view site. Whereas in multi-master replication tables are continuously updated by other master sites, materialized views are updated from one or more masters through individual batch updates, known as a **refreshes**, from a single master site or master materialized view site

Simple Views

As defined earlier simple views are created from tables and are used for creating secure manipulation over the tables or structures of the database. Views make the manipulations easier to perform on the database.

Complex Views

Complex views are by definition views of type which may comprise of many of elements, such as tables, views sequences and other similar objects of the database. When talking about the views we can have views of one table, views of one table and one view, views of multiple tables views of multiple views and so on…

Dynamic Views

Dynamic views are those types of views for which data is not stored and the expressions used to build the view are used to collect the data dynamically. These views are not executed only once when they are referred for the first time, rather they are created and the data contained in such views is updated every time the view is accessed or used in any other view or query.

Dynamic views generally are complex views, views of views, and views of multiple tables.

An example of a dynamic view creation is given below:

```
CREATE VIEW st_view1 AS (select stName, stFname, prName
FROM student
WHERE prName = 'MCS')
```

Views can be referred in SQL statements like tables

We can have view created on functions and other views as well. Where the function used for the view creation and the other nested view will be used as a simple table or relation.

Examples:

View Using another View

```
CREATE VIEW CLASSLOC2
AS SELECT COURSE#, ROOM
FROM CLASSLOC
```

View Using Function

```
CREATE VIEW CLASSCOUNT(COURSE#, TOTCOUNT)
    AS SELECT COURSE#,      COUNT(*)
    FROM ENROLL
    GROUP BY COURSE#;
```

Dynamic Views

SELECT * FROM st_view1

	stName	stFname	prName
1	Amjad	Hussain	MCS
2	Amjad	Rehan	MCS
3	Tahira Ejaz	Nek Muhammad	MCS
4	Suhal Dar	Loving	MCS

With Check Option

```
CREATE VIEW st_view2
AS (SELECT stName, stFname, prName FROM student WHERE prName = 'BCS')
WITH CHECK OPTION
```

```
UPDATE ST_VIEW1 set prName = 'BCS'
Where stFname = 'Loving'
```

SELECT * from ST_VIEW1

	stName	stFname	prName
1	Amjad	Hussain	MCS
2	Amjad	Rehan	MCS
3	Tahira Ejaz	Nek Muhammad	MCS

SELECT * FROM ST_VIEW2

	stName	stFname	prName
1	Suhal Dar	Loving	BCS
2	Shoaib Ali	Rahmat Ali	BCS

Update ST_VIEW2 set prName = 'MCS'
Where stFname = 'Loving'

```
Server: Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that
The statement has been terminated.
```

Characteristics of Views

- Computed attributes
- Nesting of views

CREATE VIEW enr_view AS (select * from enroll)

CREATE VIEW enr_view1 as (select stId, crcode, smrks, mterm, smrks + mterm sessional from enr_view)

Select * from enr_view1

	stId	crcode	smrks	mterm	sessional
1	S1015	CS-516	12	32	44
2	S1015	CS-616	10	30	40
3	S1018	MG-314	10	26	36
4	S1020	CS-516	13	26	39
5	S1020	CS-616	12	25	37

Deleting Views:

A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

Updates on Views

Updating a view is quite simple and is performed in the same way as we perform updates on any of the database relations. But this simplicity is limited to those views only which are created using a single relation. Those views which comprise of multiple relations the updation are hard to perform and needs additional care and precaution.

As we know that the views may contain some fields which are not the actual data fields in the relation but may also contain computed attributes. So update or insertions in this case are not performed through the views created on these tables.

Lecture No. 41

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture

- Indexes
- Index Classification

In our previous lecture we were discussing the views. Views play an important role in database. At this layer database is available to the users. The user needs to know that they are dealing with views; it is infact virtual for them. It can be used to implement security. We were discussing dynamic views whose data is not stored as such.

Updating Multiple Tables

We can do this updation of multiple views by doing it one at a time. It means that while inserting values in different tables, it can only be done one at a time. We will now see an example of this as under:

CREATE VIEW st_pr_view1 (a1, a2, a3, a4) AS (select stId, stName, program.prName, prcredits from student, program WHERE student.prName = program.prName)
In this example this is a join statement

We will now enter data in the table

insert into st_pr_view1 (a3, a4) values ('MSE', 110)
We will now see the program table after writing this SQL statement as the data has been stored in the table.
Select * from program

	prName	totSem	prCredits
1	BBA	8	130
2	BCE	8	134
3	BCS	8	134
4	BIT	8	132
5	MBA	4	64
6	MCS	4	64
7	MIT	4	62
8	MSE	NULL	110

In this example the total semester is NULL as this attribute was not defined in view creation statement, so then this value will remain NULL. We will now see another example. In this we have catered for NOT NULL.
insert into st_pr_view1 (a1, a2) values ('S1043', 'Bilal Masood')

SELECT * from student

	stId	stName	stFName	stAdres	stPhone	prName	curSem
1	S0123	Amjad	Hussain	8-SD Lahore	234322	MCS	1
2	S1012	Amjad	Rehan	I8 Ibd	5456754	MCS	2
3	S1015	Tahira Ejaz	Nek Muhammad	AJ Road	4323456	MCS	3
4	S1018	Arif Zia	Zia Khan	GM Rawalpindi	4356488	BIT	2
5	S1020	Suhal Dar	Loving	I-8 Islamabad	5523240	BCS	2
6	S1021	M. Ali	M. Saad	JT Lahore	544325	MBA	2
7	S1034	Sadia Zia	NULL	NULL	NULL	BIT	1
8	S1038	Shoaib Ali	Rahmat Ali	G-6 Islamabad	5343240	BCS	2
9	S1040	Ahmad Ali	Ali Hussain	NULL	NULL	BCS	1
10	S1042	Ahmad Ali	Ali Hussain	NULL	NULL	BCS	1
11	S1044	Bilal Moasood	NULL	NULL	NULL	NULL	1

Materialized Views

A pre-computed table comprising aggregated or joined data from fact and possibly dimensions tables. Also known as summary or aggregate table.Views are virtual tables. In which query is executed every time .For complex queries involving large number of join rows and aggregate functions, so it is problematic. Its solution is materialized views also called indexed views created through clustered index. Creating a clustered index on a view stores the result set built at the time the index is created. An indexed view also automatically reflects modifications made to the data in

the base tables after the index is created, the same way an index created on a base table does. Create indexes only on views where the improved speed in retrieving results outweighs the increased overhead of making modifications.

Materialized views are schema objects that can be used to summarize, compute, replicate, and distribute data. They are suitable in various computing environments such as data warehousing, decision support, and distributed or mobile computing:

- In data warehouses, materialized views are used to compute and store aggregated data such as sums and averages. Materialized views in these environments are typically referred to as summaries because they store summarized data. They can also be used to compute joins with or without aggregations. If compatibility is set to Oracle9*i* or higher, then materialized views can be used for queries that include filter selections.

  Cost-based optimization can use materialized views to improve query performance by automatically recognizing when a materialized view can and should be used to satisfy a request. The optimizer transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying detail tables or views.

- In distributed environments, materialized views are used to replicate data at distributed sites and synchronize updates done at several sites with conflict resolution methods. The materialized views as replicas provide local access to data that otherwise have to be accessed from remote sites.
- In mobile computing environments, materialized views are used to download a subset of data from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients back to the central servers.

Materialized views are similar to indexes in several ways:

- They consume storage space.
- They must be refreshed when the data in their master tables changes.

- They improve the performance of SQL execution when they are used for query rewrites.
- Their existence is transparent to SQL applications and users.

Unlike indexes, materialized views can be accessed directly using a SELECT statement. Depending on the types of refresh that are required, they can also be accessed directly in an INSERT, UPDATE, or DELETE statement.
A materialized view can be partitioned. You can define a materialized view on a partitioned table and one or more indexes on the materialized view.

Transaction Management

A transaction can be defined as an indivisible unit of work comprised of several operations, all or none of which must be performed in order to preserve data integrity. For example, a transfer of Rs 1000 from your checking account to your savings account would consist of two steps: debiting your checking account by Rs1000 and crediting your savings account with Rs1000. To protect data integrity and consistency and the interests of the bank and the customer these two operations must be applied together or not at all. Thus, they constitute a transaction.

Properties of a transaction

All transactions share these properties: atomicity, consistency, isolation, and durability (represented by the acronym ACID).

- Atomicity: This implies indivisibility; any indivisible operation (one which will either complete fully or not at all) is said to be atomic.
- Consistency: A transaction must transition persistent data from one consistent state to another. If a failure occurs during processing, the data must be restored to the state it was in prior to the transaction.
- Isolation: Transactions should not affect each other. A transaction in progress, not yet committed or rolled back (these terms are explained at the end of this section), must be isolated from other transactions. Although several transactions may run concurrently, it should appear to each that all the others

completed before or after it; all such concurrent transactions must effectively end in sequential order.

- Durability: Once a transaction has successfully committed, state changes committed by that transaction must be durable and persistent, despite any failures that occur afterwards.

A transaction can thus end in two ways: a commit, the successful execution of each step in the transaction, or a rollback, which guarantees that none of the steps are executed due to an error in one of those steps.

Lecture No. 42

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.
"Database Management System" by Jeffery A Hoffer

Overview of Lecture

- Transaction Management
- The Concept of a Transaction
- Transactions and Schedules
- Concurrent Execution of Transactions
- Need for Concurrency Control
- Serializability
- Locking
- Deadlock

A transaction is defined as any one execution of a user program in a DBMS and differs from an execution of a program outside the DBMS (e.g., a C program executing on Unix) in important ways. (Executing the same program several times will generate several transactions.)

For performance reasons, a DBMS has to interleave the actions of several transactions. However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect upon the database) to some serial, or one-at-a-time, execution of the same set of transactions.

The Concept of a Transaction

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard an execution of a user program, or transaction, as a series of reads and writes of database objects:

- To read a database object, it is first brought into main memory (specifically, some frame in the buffer pool) from disk, and then its value is copied into a program variable.
- To write a database object, an in-memory copy of the object is first modified and then written to disk.

Database `objects' are the units in which programs read or write information. The units could be pages, records, and so on, but this is dependent on the DBMS and is not central to the principles underlying concurrency control or recovery. In this chapter, we will consider a database to be a fixed collection of independent objects.

There are four important properties of transactions that a DBMS must ensure to maintain data in the face of concurrent access and system failures:

- Users should be able to regard the execution of each transaction as atomic: either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).
- Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. This property is called consistency, and the DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
- Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
- Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.

The acronym ACID is sometimes used to refer to the four properties of transactions which are presented here: atomicity, consistency, isolation and durability.

Transactions and Schedules

A transaction is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects. A transaction can also be defined as a set of actions that are partially ordered. That is, the relative order of some of the actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a list of actions. Further, to keep our notation simple, we'll assume that an object O is always read into a program variable that is also named O. We can therefore denote the action of a transaction T reading an object O as RT (O); similarly, we can denote writing as WT (O). When the transaction T is clear from the context, we will omit the subscript.

A schedule is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure below shows an execution order for actions of two transactions T1 and T2. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions as seen by the DBMS. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on.

Concurrent Execution of Transactions

The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleaving should be allowed.

Ensuring transaction isolation while permitting such concurrent execution is difficult but is necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases system throughput (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short

transaction could get stuck behind a long transaction leading to unpredictable delays in response time, or average time taken to complete a transaction.

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure: Transaction Scheduling

Serializability

To begin with, we assume that the database designer has defined some notion of a consistent database state. For example, we can define a consistency criterion for a university database to be that the sum of employee salaries in each department should be less than 80 percent of the budget for that department. We require that each transaction must preserve database consistency; it follows that any serial schedule that is complete will also preserve database consistency. That is, when a complete serial schedule is executed against a consistent database, the result is also a consistent database.

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order. There are some important points to note in this definition:

- Executing the transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- The above definition of a serializable schedule does not cover the case of schedules containing aborted transactions
- If a transaction computes a value and prints it to the screen, this is an `effect' that is not directly captured in the state of the database. We will assume that all such values are also written into the database, for simplicity.

Lock-Based Concurrency Control

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a locking protocol to achieve this. A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules.

The first rule is

If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object. Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is:

All locks held by a transaction are released when the transaction is completed. Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details. In effect the locking protocol allows only `safe' interleaving of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as ST (O) (respectively, XT (O)), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure below.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Figure reading uncommitted Data

This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, T1 could change A from 10 to 20, then T2 (which reads the value 20 for A) could change B from 100 to 200, and then T1 would read the value 200 for B. If run serially, either T1 orT2 would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution. If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, T1 would obtain an exclusive lock on A first and then read and write A (figure below) Then, T2 would request a lock on A. However, this request cannot be granted until T1 releases its exclusive lock on A, and the DBMS therefore suspends T2. T1 now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. T2's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure below, which is permitted by the Strict 2PL protocol.

T1	T2
X(A)	
R(A)	
W(A)	

Figure: Schedule illustrating strict 2PL.

Deadlocks

Consider the following example: transaction T1 gets an exclusive lock on object A, T2 gets an exclusive lock on B, T1 requests an exclusive lock on B and is queued, and T2 requests an exclusive lock on A and is queued. Now, T1 is waiting for T2 to release its lock and T2 is waiting for T1 to release its lock! Such a cycle of transactions waiting for locks to be released is called a deadlock. Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations.

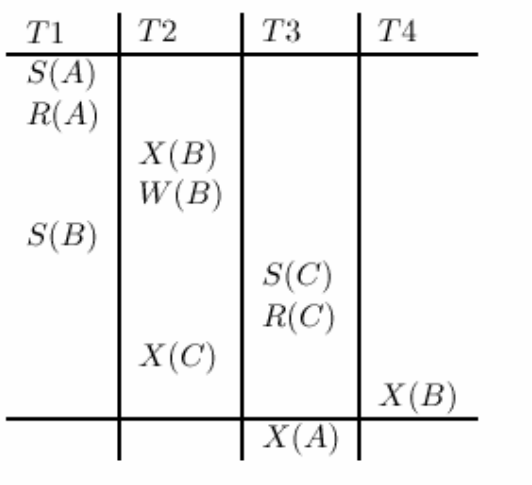


Figure :Schedule Illustrating Deadlock

Deadlock Prevention

We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher the transaction's priority, that is, the oldest transaction has the highest priority.

If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:

- Wait-die: If Ti has higher priority, it is allowed to wait; otherwise it is aborted.
- Wound-wait: If Ti has higher priority, abort Tj; otherwise Ti waits.

In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

A subtle point is that we must also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that in both schemes, the higher priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp that it had originally. Reissuing timestamps in this way ensures that each transaction will eventually become the oldest transaction, and thus the one with the highest priority, and will get all the locks that it requires.

The wait-die scheme is non-preemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more young transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound wait), but on the other hand, a transaction that has all the locks it needs will never be aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

Deadlock Detection

Deadlocks tend to be rare and typically involve very few transactions. This observation suggests that rather than taking measures to prevent deadlocks, it may be better to detect and resolve deadlocks as they arise. In the detection approach, the DBMS must periodically check for deadlocks. When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them.

The lock manager maintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from Ti to Tj if (and only if) Ti is waiting for Tj to release a lock. The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.

Detection versus Prevention

In prevention-based schemes, the abort mechanism is used preemptively in order to avoid deadlocks. On the other hand, in detection-based schemes, the transactions in a deadlock cycle hold locks that prevent other transactions from making progress. System throughput is reduced because many transactions may be blocked, waiting to obtain locks currently held by deadlocked transactions.

This is the fundamental trade-off between these prevention and detection approaches to deadlocks: loss of work due to preemptive aborts versus loss of work due to

blocked transactions in a deadlock cycle. We can increase the frequency with which we check for deadlock cycles, and thereby reduce the amount of work lost due to blocked transactions, but this entails a corresponding increase in the cost of the deadlock detection mechanism.

A variant of 2PL called Conservative 2PL can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks that it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will not be any deadlocks, and, perhaps more importantly, that a transaction that already holds some locks will not block waiting for other locks. The trade-o_ is that a transaction acquires locks earlier. If lock contention is low, locks are held longer under Conservative 2PL. If lock contention is heavy, on the other hand, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked.

Lecture No. 43

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.	Chapter 12
"Database Management Systems" Second edition written by Raghu Ramakrishnan, Johannes Gehkre.	Chapter 18

Overview of Lecture

- Incremental log with deferred updates
- Incremental log with immediate updates
- Introduction to concurrency control

Incremental Log with Deferred Updates

We are discussing the deferred updates approach regarding the database recovery techniques. In the previous lecture we studied the structure of log file entries for the deferred updates approach. In today's lecture we will discuss the recovery process.

Write Sequence:
First we see what the sequence of actions is when a write operation is performed. On encountering a 'write' operation, the DBMS places an entry in the log file buffer mentioning the effect of the write operation. For example, if the transaction includes the operations:

…….
X = X + 10
Write X
…….

Supposing that the value of X before the addition operation is 23 and after the execution of operation it becomes 33. Now against the write operation, the entry made in the log file will be

<Tn, X, 33>

In the entry, Tn reflects the identity of the transaction, X is the object being updated and 33 is the value that has to be placed in X. Important point to be noted here is that the log entry is made only for the write operation. The assignment operation and any other mathematical or relational operation is executed in RAM. For a 'read' operation the value may be read from the disk (database) or it may already be in RAM or cache. But the log entry results only from a 'write' operation.

The transaction proceeds in a similar fashion and entries keep on adding for the write operations. When the 'commit' statement is executed then, first, the database buffer is updated, that is the new value is placed in the database buffer. After that the log file is moved to disk log file which means that the log file entries have been made permanent. Then write is performed, that is, the data from DB buffer is moved to disk. There may be some delay between the execution of the commit statement and the actual writing of data from database buffer to disk. Now, during this time if system crashes then the problem is that the transaction has been declared as committed whereas the final result of the object was still in RAM (in database buffer) and due to system crash the contents of the RAM have been lost. The crash recovery procedure takes care of this situation.

The Recovery Procedure

When crash occurs, the recovery manager (RM), on restart, examines the log file from the disk. The transactions for which the RM finds both begin and commit operations, are redone, that is, the 'write' entries for such transactions are executed again. For example, consider the example entries in a log file

Log File Entries
<T1, begin >
<T1, X, 33>
<T2, begin>
<T1, Y, 9>
<T2, A, 43>
<T1, X, 18>
<T1, commit >
<T2, abort>
<T3, begin>
<T3, B, 12>

Here we find the begin entries for T1, T2 and T3, however, the commit entry is only for T1. On restart after the crash, the RM performs the write operations of T1 again in the forward order, that is, it writes the value 33 for object X, then writes 9 for Y and writes 19 for X again. If some or all of these operations have been performed already, writing them again will not cause any harm.

The RM does not any action for the transactions for which there is only a begin entry or for which there are begin and abort entries, like T3 and T2 respectively.
A question arises here, that how far should the RM go backward in the log file to trace the transactions affected from the crash that the RM needs to Redo or ignore. The log file may contain so many entries and going too much back or starting right from the start of the log file will be inefficient. For this purpose another concept of 'checkpoint' is used in the recovery procedure.

Checkpoint:
Checkpoint is also a record or an entry in the log file. It serves as a milestone or reference point in the log file. At certain point in time the DBMS enters a log entry/record in the log file and also performs certain operations listed below:

- **It moves modified database buffers to disk**
- **All log records from buffer to disk**
- **Writing a checkpoint record to log; log record mentions all active transactions at the time**

Now in case of crash, the RM monitors the log file up to the last checkpoint. The checkpoint guarantees that any prior commit has been reflected in the database. The RM has to decide which transactions to Redo and which to ignore and RM decides it on the following bases:

- **Transactions ended before the checkpoint, are ignored altogether.**
- **Transactions which have both begin and the commit entries, are Redone. It does not matter if they are committed again.**
- **Transactions that have begin and an abort entry are ignored.**
- **Transactions that have a begin and no end entry (commit or rollback) are ignored**

This way checkpoint makes the recovery procedure efficient. We can summarize the deferred approach as:

- **Writes are deferred until commit for transaction is found**
- **For recovery purpose, log file is maintained**
- **Log file helps to redo the actions that may be lost due to system crash**
- **Log file also contains checkpoint records**

Next, we are going to discuss the other approach of crash recovery and that is incremental log with immediate updates.

Incremental Log with Immediate Updates

Contrary to the deferred updates, the database buffers are updated immediately as the 'write' statement is executed and files are updated when convenient. The log file and database buffers are maintained as in the deferred update approach. One effect of immediate update is that the object updating process needs not to wait for the commit statement; however, there is a problem that the transaction in which a 'write' statement has been executed (and the buffer has been updated accordingly) may be aborted later. Now there needs to be some process that cancels the updation that has been performed for the aborted transaction. This is achieved by a slightly different log file entry/record. The structure of log file entry for immediate update technique is <Tr, object, old_value, new_value>, where Tr represents the transaction Id, 'object' is the object to be updated, old_value represents the value of object before the execution of 'write' statement and new_value is the new value of object. Other entries of the log file in immediate update approach are just like those of deferred update.
The sequence of steps on write statement is:

- **A log record of the form <T, X, old val, new val> is written in the log file**
- **Database buffers are updated**
- **Log record is moved from buffer to the file**
- **Database is updated when convenient**
- **On commit a log file entry is made in the log file as <T, commit>**

Recovery Sequence

In case of a crash, the RM detects the crash as before on restart and after that the recovery procedure is initiated. The transactions for which the <Tr, begin> and <Tr, commit> are found, those transactions are redone. The redo process is performed by copying the new value of the objects in the forward order. If these transactions have already been successfully executed even then this redo will not do any harm and the final state of the objects will be the same. The transactions for which a <Tr, begin> and <Tr, abort> are found or those transaction for which only <Ts, begin> is found, such transactions are undone. The undo activity is performed by executing the write statements in the reverse order till the start of the transaction. During this execution, the old value of the objects are copied/placed so all the changes made by this transaction are cancelled.

Log File Entries
<T1, begin>
<T1, X, 25, 33>
<T2, begin>
<T1, Y, 5, 9>
<T2, A, 80, 43>
<T1, X, 33, 18>
<T1, commit>
<T3 begin >
<T2, A, 43, 10>
<T3, B, 12, 9>
<T2, abort>

Here we find the begin entries for T1, T2 and T3, however, the commit entry is only for T1 and also there is an abort entry for T2. On restart after the crash, the RM performs the write operations of T1 again in the forward order, that is, it writes the value 33 for object X, then writes 9 for Y and writes 18 for X again. If some or all of these operations have been performed already, writing them again will not cause any harm.

To undo the effects of T2 (which has been aborted) the write statements of T2 are executed in the reverse order copying the old value from the entry/record. Like, here the second write statement of T2 that had possibly written the value 10 over 43, now 43 will be placed for A. Moving backward we find another write statement of T2 that places 43 replacing 80. Now we pick the old value, i.e., 80 and place this value for A. After performing these operations the effects of executing an aborted T2 are completely eliminated, that is, the object A contains value 80 that it had before the execution of transaction T2. Same process is also applied for the transaction T3. Checkpoints are used in immediate updates approach as well.

We have discussed recovery procedure both in deferred update and immediate update. Crash recovery is an important aspect of a DBMS, since in spite of all precautions and protections the crashes happen. We can minimize the frequency but they cannot be avoided altogether. Once crash happens, the purpose of recovery mechanism is to

reduce the effect of this crash as much as possible and to bring database in a consistent state and we have studied how the DBMS performs this activity.

Concurrency Control

Concurrency control (CC) is the second part of the transaction management. The objective of the CC is to control the concurrent access of database by multiple users at the same time called the concurrent access.

First questions is that why to have the concurrent access. The answer to this question refers to very basic objective of the database approach that is sharing of data. Now if we are allowing multiple users to share the data stored in the database, then how will they access this data. One user at a time? If so, it will be very inefficient, we cannot expect the users to be that patient to wait so long. At the same time we have so powerful and sophisticated computer hardware and software that it will be a very bad under-utilization of the resources if allow one user access at one time. That is why concurrent access of the data is very much required and essential.

We are convinced that concurrent access is required, but if not controlled properly the concurrent access may cause certain problems and those problems may turn the database into an inconsistent state and this is never every allowed in the database processing. The objective of CC is to allow the concurrent access in such a way that these problems due to concurrent access are not encountered or to maintain the consistency of the database in the concurrent access. It is important to understand that CC only takes care of the inconsistencies that may possibly be encountered due to concurrent access. Inconsistency due to some other reasons is not the concern of CC.

Problems due to Concurrent Access

If not controlled properly, the concurrent access may introduce following three problems in database:

- **Lost Update Problem**
- **Uncommitted Update Problem**
- **Inconsistent Analysis Problem**

We discuss these problems one by one

Lost Update Problem

This problem occurs when multiple users want to update same object at the same time. This phenomenon is shown in the table below:

TIME	TA	TB	BAL
t_1	Read (BAL)		1000
t_2	…….	Read (BAL)	1000
t_3	BAL = BAL - 50	……..	1000
t_4	Write (BAL)		950
t_5	…….	BAL = BAL + 10	950
t_6	……..	Write (BAL)	1010

Table 1: Lost update problem

This first column represents the time that acts as reference for the execution of the statements. As is shown in the table, at time t1 transaction TA reads the value of object 'BAL' that supposedly be 1000. At time t2, transaction TB reads the value of same object 'BAL' and also finds the value as 1000. Now at time t3, TA subtracts 50 of 'BAL and writes it at time t4. On the other hand TB adds 10 into value of 'BAL' (that she has already read as 1000) and writes the value of 'BAL'. Now since TB wrote the value after TA, the update made by TA is lost, it has been overwritten by value of TB. This is the situation that reflects the lost update problem.

This problem occurred because concurrent access to the same object was not controlled properly, that is, concurrency control did not manage the things properly. There are two more situations that reflect the problems of concurrent access that we will discuss in the next lecture.

Summary

This lecture concluded the discussion on crash recovery where we studied that the main tool used for recovery is the log file. The structure of records stored in the different types of log files is almost the same, except for the record that stores the record of the change made in the database as a result of a write statement. The deferred update approach stores only the new value of the object in the log file. Whereas the immediate update approach stores the previous as well as new value of the object being updated. After the crash recovery we started the discussion on CC,

where we were studying different problems of CC; first of them is the lost update problem that we have discussed in today's lecture, rest two will be discussed in the next lecture.

Lecture No. 44

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.	Chapter 12
"Database Management Systems" Second edition written by Raghu Ramakrishnan, Johannes Gehkre.	Chapter 18

Overview of Lecture

- Concurrency control problems
- Serial and interleaved schedules
- Serializability theory
- Introduction to locking

We are studying the concurrency control (CC) mechanism. In the previous lecture we discussed that concurrent access means the simultaneous access of data from database by multiple users. The concurrency control (CC) concerns maintaining the consistency of the database during concurrent access. We also studied that if concurrent access is not controlled properly then database may encounter three different problems which may turn database into an inconsistence state. One of these problems we discussed in the previous lecture, now we will discuss the remaining two.

Uncommitted Update Problem

It reflects a situation when a transaction updates an object and another transaction reads this updated value but later the first transaction is aborted. The problem in this situation is that the second transaction reads the value updated by an aborted transaction. This situation is shown in the table below:

TIME	TA	TB	BAL
t_1	Read (BAL)	…….	1000
t_2	BAL=BAL+100	……..	1000
t_3	Write (BAL)	……..	1100
t_4	…….	Read (BAL)	1100
t_5	…….	BAL=BAL*1.5	1100
t_6	…….	Write (BAL)	2650
t_7	ROLLBACK	……..	?

Table 1: Uncommitted update problem

As is shown in the table, at time t2, transaction TA updates the value of object 'BAL' by adding 100 in its initial value 1000 and at time t3 it writes this updated value to database. So 'BAL' is written as 1100. Now at time t4, TB reads the value of 'BAL' and it gets 1100 then TB updates the value multiplying it by 1.5 and writes it at time t6. So the new value written for 'BAL' is 2650. However at time t7 transaction TA is rolled back as a result of this rollback the changes made by TA stand cancelled, but TB has already performed certain processing on the value set by TA. This is an inconsistent state of the database.

Inconsistent Analysis

This problem of concurrent access occurs when one transaction operates on many records, meanwhile another modifies some of them effecting result of first. This problem is expressed in the following table:

TIME	TA	TB
t_1	Read (BAL_A) (5000)	…….
t_2	INT = BAL_A * .05	……..
t_3	TOT = TOT + INT	……..
t_4	…….	…….
t_5	…….	Read (BAL_A)
t_6	…….	BAL_A=BAL_A – 1000
t_7	…….	Write (BAL_A)
t_8	…….	Read (BAL_E) (5000)
t_9	…….	BAL_E = BAL_E + 1000
t_{10}	…….	Write (BAL_E)
t_{11}	Read (BAL_E) (6000)	……..
t_{12}	INT = BAS_E * .05	……..
t_{13}	TOT = TOT + INT	……..

Table 2: Inconsistent analysis problem

Suppose Tr-A computes interest on all accounts balances. For this, TA reads the balance of each account multiplies it with 0.05 and adds this interest amount into a variable TOT. Now by time t5, TA has read the balance of account ‘A’ (BAL_A) and computed interest on it, it has to perform same process on all the accounts. In the meanwhile from time t5 to t10 another transaction TB subtracts a value from balance of account ‘A’ and adds this value to the balance of account ‘E’. When transaction TA reaches the account ‘E’ to compute interest on it, the value added in it by TB is also considered which as a matter of fact is being considered second time. First time from account ‘A’ and now from account ‘E’. The analysis obtained through transaction TA will be wrong at the end.

We have discussed three problems of concurrent access; the CC mechanism has to make sure that these problems do not occur in the database. In the following, we will discuss how it is done. We will start by studying some basic concepts.

Serial Execution

Serial execution is an execution where transactions are executed in a sequential order, that is, one after another. A transaction may consist of many operations. Serial execution means that all the operations of one transaction are executer first, followed by all the operations of the next transaction and like that. A Schedule or History is a list of operations from one or more transactions. A schedule represents the order of execution of operations. The order of operations in a schedule should be the same as in the transaction. Schedule for a serial execution is called a serial schedule, so in a serial schedule all operations of one transactions are listed followed by all the

operations of another transactions and so on. With a given set of transactions, we can have different serial schedules. For example, if we have two transactions, then we can have two different serial schedules as is explained in the table below:

Serial Sched 1 TA, TB	BAL	Serial Sched 2 TB, TA	BAL
Read $(BAL)_A$	50	Read $(BAL)_B$	50
$(BAL = BAL – 10)_A$	50	$(BAL=BAL * 2)_B$	50
Write $(BAL)_A$	40	Write $(BAL)_B$	100
Read $(BAL)_B$	40	Read $(BAL)_A$	100
$(BAL=BAL * 2)_B$	80	$(BAL = BAL – 10)_A$	100
Write $(BAL)_B$	80	Write $(BAL)_A$	90

Table 3: Two different serial schedules, TA, TB and TB, TA

The table shows two different schedules of two transactions TA and TB. The subscript with each operation shows the transaction to which the operation belongs. For example, Read $(BAL)_A$ means the read operation of TA to read the object 'BAL'. By looking at the table that in each serial schedule; all the operations of one transaction are executed first followed by those of the other transaction. An important point to be noted here is that different serial schedules of the same transactions may result in different final state of the database. As we can see in the table 3, final value of the object 'BAL' is 80 with the serial schedule TA, TB and it is 90 with serial schedule TB, TA. However, it is guaranteed that a serial schedule always leaves the database in a consistent state. In other words, the three problems of concurrency that we studies earlier will not occur if a serial schedule is followed.

The serial schedule ensures the consistency of the database, so should we prefer serial schedule? The answer is no. Because serial execution is badly under utilization of resources and users will not like it at all since they will have to wait a lot for the transactions' execution. Suppose we are following a serial execution and a transaction Tn has to executed completely before any other transaction may start. Lets say Tn needs an input from the user who initiated it, and by chance users gets stuck somewhere or is thinking, unless and until the users does not enter a value, transaction Tn cannot proceed and all other transactions are waiting for their turn to execute. If

allowed to happen, this will be a much disliked situation. Therefore serial schedules are not preferred for execution.

Contrary to serial schedule is an interleaved schedule in which transactions' execution is interleaved, that is, operations of different transactions are intermix with each other. However, the internal order of the operations is not changed during this intermixing. Interleave schedule provides more efficient execution since control is switched among transaction, so on one side it makes a better use of the resources and on the other hand if one transaction is stuck or halted due to some reasons, the processing can go on with other transactions. Following figure presents serial and interleaved schedules among two transactions TA and TB

TA = {Read(X), Write(X), commit}
TB = {Read(Y), Read(X), Write(Y), commit}

$S_1 = \{R_A(X), W_A(X), C_A, R_B(Y), R_B(X), W_B(Y), C_B\}$

$S_2 = \{R_B(Y), R_B(X), W_B(Y), C_B, R_A(X), W_A(X), C_A\}$

$S_3 = \{R_B(Y), R_B(X), R_A(X), W_B(Y), C_B, W_A(X), C_A\}$

$S_4 = \{R_A(X), R_B(Y), R_B(X), W_A(X), C_A, W_B(Y), C_B\}$

Fig. 1: Two transactions and different schedules

In figure 1, we have two transactions TA and TB consisting of different operations. S1, S2, S3 and S4 are four different schedules of these transactions. S1 and S2 are two serial schedules where as S3 and S4 are two interleaved schedules. Obviously we can have just two serial schedules of two transactions, but we can have many other interleaved schedules of these transactions.

Interleaved schedules are preferred but they may generate three of the concurrent access problems if not controlled properly. Before discussing the solution to this problem, lets see what is the actual problem in concurrent access.

There are different situations during concurrent access of the data:

- Different transactions accessing (reading or writing) different objects
- Different transactions reading same object

- Different transactions accessing same object and one or all writing it

The first two of these situations do not create any problem during concurrent access, so we do not need to worry in these situations. However, third situation is precisely the one that creates the concurrency problems and we have to control this situation properly. Third situation introduces the concept of conflicting operations; the operations that are accessing the same object and one of them writing the object. The transactions that contain conflicting operations are called conflicting transactions. Basically, in concurrency control we have to take care of the conflicting transactions and for that we have to study the concept of serializability.

Serializability

An interleaved schedule is said to be serializable if its final state is equivalent to some serial schedule. Serializable schedule can also be defined as a schedule in which conflicting operations are in a serial order. The serializable schedule ensures the consistency of the database. However, this is worthwhile to mention here again that serializability takes care of the inconsistencies only due to the interleaving of transactions. The serializability concerns generating serializable schedules or we can say that the purpose of the CC mechanism is to ensure serializability of the schedules. Generating serializable schedule involves taking care of only conflicting operations. We have to adopt a particular serial schedule among the conflicting operations; the non-conflicting operations can be interleaved to any order, it will not create any problem at all. There are two major approaches to implement the serializability; Locking and Timestamping. We will discuss both in detail.

Locking

The basic idea of locking is that an object is locked prior to performing any operation. When the operation has been performed the lock is released. Locking is maintained by the transaction manager or more precisely lock manager. Transactions apply locks on objects. During the time when an object is locked it is available for operations only to the transaction that has got the lock on that object through lock manager. When an item is locked and during that time another transaction needs to perform some

operation on that object, the transaction applies for the lock on the object but since the object is already locked, this transaction will have to wait. So it enters into a wait state. When first transaction finishes its job, it releases locks on items then the second item gets the lock on the object and then it can proceed.

Transactions perform two types of operations on objects; read or write. That is, a transaction either performs a read operation on an object or it writes it. Accordingly there are two types of locks as well. A read or shared lock and a write or exclusive lock a transaction applies lock according to the nature of operation that it wants to perform on a particular object. If a transaction TB applies a lock on an object O, the lock manager first checks if O is free, it is free then TB gets the desired lock on O. However, if O is already locked by another transaction say TA then lock manager checks the compatibility of the locks; the one that has already been assigned and the one that has been applied now. The compatibility of locks means that if the two locks from two different transactions may exist at the same time. The compatibility of locks is checked according to the following table:

Transaction A

Transaction B		Read	Write
	Read	*Yes*	*No*
	Write	*No*	*No*

Table shows that if a transaction A has got a read lock on an object and transaction B applies for the read lock, B will be granted this lock. It means two read locks are compatible with each other, that is, they can exist at the same time. Therefore two or even more than two transactions can read an object at the same time. Other cells contain 'No'; it means these locks are not compatible. So if a transaction has got a 'write' lock on an object and another transaction applies for a 'read' or 'write' lock, the lock will not be granted and the transaction applying for the lock later, will have to wait.

That is all for today's lecture. The locking mechanism will be discussed in detail in the next lecture.

Summary

In this lecture we discussed two of the three CC problems. Then we discussed that there are different types of schedules that determine the sequence of execution of

operations form different transactions. A serial schedule maintains the consistency of the database for sure, but serial schedule is not much useful. Interleaved schedule is preferred but if not controlled properly an interleaved schedule may cause consistency problems. So CC is based on the serializability theory that controls the order of conflicting operations. The serializability is applied using locking or Timestamping. Locking is based on two types of locks; shared and exclusive. Only two shared locks are compatible with each other, none of the remaining combinations are compatible.

Lecture No. 45

Reading Material

"Database Systems Principles, Design and Implementation" written by Catherine Ricardo, Maxwell Macmillan.

Overview of Lecture:

- Deadlock Handling
- Two Phase Locking
- Levels of Locking
- Timestamping

This is the last lecture of our course; we had started with transaction management. We were discussing the problems in transaction in which we discussed the locking mechanism. A transaction may be thought of as an interaction with the system, resulting in a change to the system state. While the interaction is in the process of changing system state, any number of events can interrupt the interaction, leaving the state change incomplete and the system state in an inconsistent, undesirable form. Any change to system state within a transaction boundary, therefore, has to ensure that the change leaves the system in a stable and consistent state.

Locking Idea

Traditionally, transaction isolation levels are achieved by taking locks on the data that they access until the transaction completes. There are two primary modes for taking locks: optimistic and pessimistic. These two modes are necessitated by the fact that when a transaction accesses data, its intention to change (or not change) the data may not be readily apparent.

Some systems take a pessimistic approach and lock the data so that other transactions may read but not update the data accessed by the first transaction until the first transaction completes. Pessimistic locking guarantees that the first transaction can always apply a change to the data it first accessed.

In an optimistic locking mode, the first transaction accesses data but does not take a lock on it. A second transaction may change the data while the first transaction is in progress. If the first transaction later decides to change the data it accessed, it has to detect the fact that the data is now changed and inform the initiator of the fact. In optimistic locking, therefore, the fact that a transaction accessed data first does not guarantee that it can, at a later stage, update it.

At the most fundamental level, locks can be classified into (in increasingly restrictive order) shared, update, and exclusive locks. A shared lock signifies that another transaction can take an update or another shared lock on the same piece of data. Shared locks are used when data is read (usually in pessimistic locking mode).

An update lock ensures that another transaction can take only a shared lock on the same data. Update locks are held by transactions that intend to change data (not just read it). If a transaction locks a piece of data with an exclusive lock, no other transaction may take a lock on the data. For example, a transaction with an isolation level of read uncommitted does not result in any locks on the data read by the transaction, and a transaction with repeatable read isolation can take only a share lock on data it has read.

DeadLock

A deadlock occurs when the first transaction has locks on the resources that the second transaction wants to modify, and the second transaction has locks on the resources that the first transaction intends to modify. So a deadlock is much like an infinite loop: If you let it go, it will continue until the end of time, until your server crashes, or until the power goes out (whichever comes first). Deadlock is a situation when two transactions are waiting for each other to release a lock. The transaction involved in deadlock keeps on waiting unless deadlock is over.

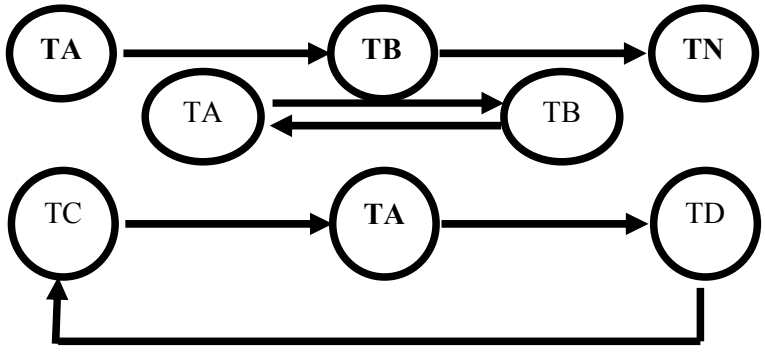
DeadLock Handling

Following are some of the approaches for the deadlock handling:

- Deadlock prevention
- Deadlock detection and resolution
- Prevention is not always possible
- Deadlock is detected by wait-for graph

Wait – for Graph:

It is used for the detection of deadlock. It consists of nodes and links. The nodes represent transaction, whereas arrowhead represents that a transaction has locked a particular data item. We will see it following example:



Now in this figure transaction A is waiting for transaction B and B is waiting for N. So it will move inversely for releasing of lock and transaction A will be the last one to execute. In the second figure there is a cycle, which represents deadlock, this kind of

cycle can be in between two or more transactions as well. The DBMS keeps on checking for the cycle.

Two Phase Locking

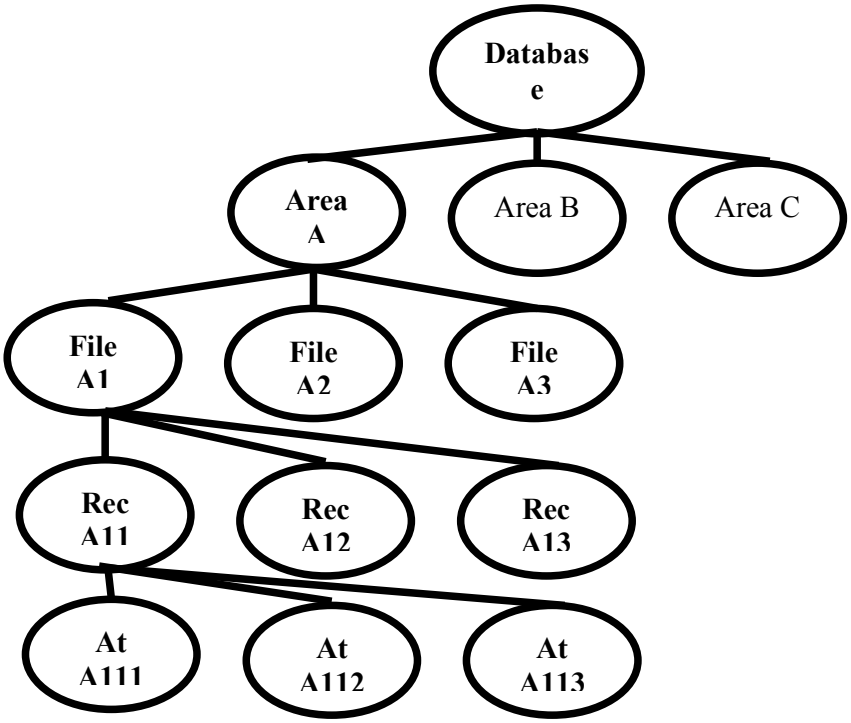
This approach locks data and assumes that a transaction is divided into a growing phase, in which locks are only acquired, and a shrinking phase, in which locks are only released. A transaction that tries to lock data that has been locked is forced to wait and may deadlock. During the first phase, the transaction only acquires locks; during the second phase, the transaction only releases locks. More formally, once a transaction releases a lock, it may not acquire any additional locks. Practically, this translates into a system in which locks are acquired as they are needed throughout a transaction and retained until the transaction ends, either by committing or aborting.
For applications, the implications of 2PL are that long-running transactions will hold locks for a long time. When designing applications, lock contention should be considered. In order to reduce the probability of deadlock and achieve the best level of concurrency possible, the following guidelines are helpful.

- When accessing multiple databases, design all transactions so that they access the files in the same order.
- If possible, access your most hotly contested resources last (so that their locks are held for the shortest time possible).
- If possible, use nested transactions to protect the parts of your transaction most likely to deadlock.

Levels of Locking

- Lock can be applied on attribute, record, file, group of files or even entire database
- Granularity of locks.
- Finer the granularity more concurrency but more overhead.

Greater lock granularity gives you finer sharing and concurrency but higher overhead; four separate degrees of consistency which give increasingly greater protection from inconsistencies are presented, requiring increasingly greater lock granularity.



When a lock at lower level is applied, compatibility is checked upward. It means intimation would be available in the hierarchy till the database. The granularity of locks in a database refers to how much of the data is locked at one time. A database server can lock as much as the entire database or as little as one column of data. Such extremes affect the concurrency (number of users that can access the data) and locking overhead (amount of work to process lock requests) in the server. By locking at higher levels of granularity, the amount of work required to obtain and manage locks is reduced. If a query needs to read or update many rows in a table:

- It can acquire just one table-level lock
- It can acquire a lock for each page that contained one of the required rows
- It can acquire a lock on each row

Less overall work is required to use a table-level lock, but large-scale locks can degrade performance, by making other users wait until locks are released. Decreasing the lock size makes more of the data accessible to other users. However, finer granularity locks can also degrade performance, since more work is necessary to maintain and coordinate the increased number of locks. To achieve optimum performance, a locking scheme must balance the needs of concurrency and overhead.

Deadlock Resolution

If a set of transactions is considered to be deadlocked:

- choose a victim (e.g. the shortest-lived transaction)
- Rollback `victim' transaction and restart it.
- The rollback terminates the transaction, undoing all its updates and releasing all of its locks.
- A message is passed to the victim and depending on the system the transaction may or may not be started again automatically.

Timestamping

Two-phase locking is not the only approach to enforcing database consistency. Another method used in some DMBS is timestamping. With timestamping, there are no locks to prevent transactions seeing uncommitted changes, and all physical updates are deferred to commit time.

- Locking synchronizes the interleaved execution of a set of transactions in such a way that it is equivalent to some serial execution of those transactions.
- Timestamping synchronizes that interleaved execution in such a way that it is equivalent to a particular serial order - the order of the timestamps.

Problems of Timestamping

- When a transaction wants to read an item that has been updated by a younger transaction.
- A transaction wants to write an item that has been read or written by a younger transaction.

Timestamping rules

The following rules are checked when transaction T attempts to change a data item. If the rule indicates ABORT, then transaction T is rolled back and aborted (and perhaps restarted).

- If T attempts to read a data item which has already been written to by a younger transaction then ABORT T.
- If T attempts to write a data item which has been seen or written to by a younger transaction then ABORT T.

If transaction T aborts, then all other transactions which have seen a data item written to by T must also abort. In addition, other aborting transactions can cause further aborts on other transactions. This is a `cascading rollback'.

Summary

In today's lecture we have read the locking mechanism and prevention of deadlocks. With this we are finished with our course. Locking is a natural part of any application. However, if the design of the applications and transactions is not done correctly, you can run into severe blocking issues that can manifest themselves in severe performance and scalability issues by resulting in contention on resources. Controlling blocking in an application is a matter of the right application design, the correct transaction architecture, a correct set of parameter settings, and testing your application under heavy load with volume data to make sure that the application scales well.